Software Model Checking:

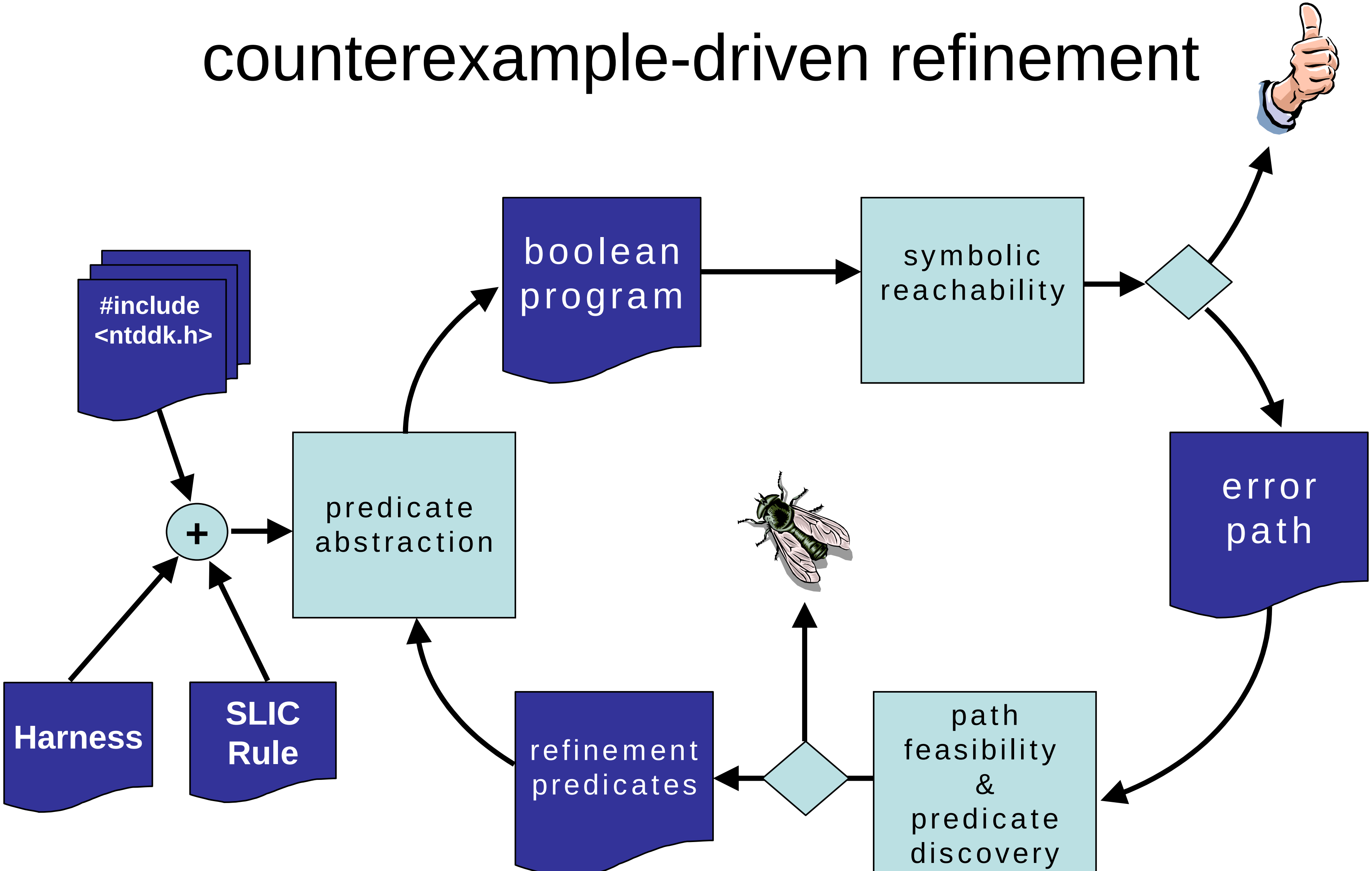
predicate abstraction

Thomas Ball
Testing, Verification and Measurement
Microsoft Research

Today and Tomorrow

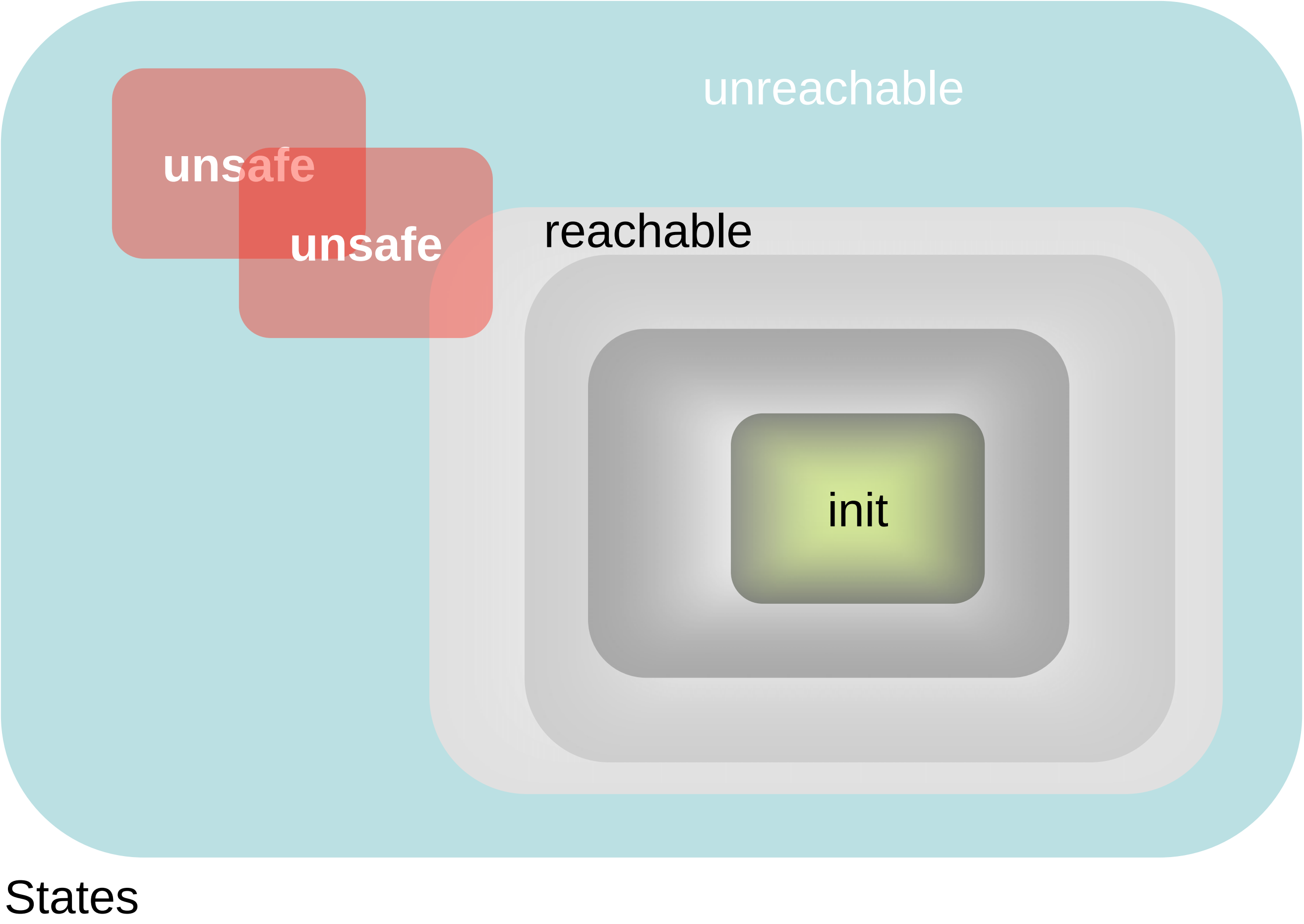
- SLAM and Static Driver Verifier Demo
- Formalizing predicate abstraction
- Predicate abstraction of programs with procedures and pointers
- Symbolic model checking of boolean programs

The SLAM Process:
counterexample-driven refinement



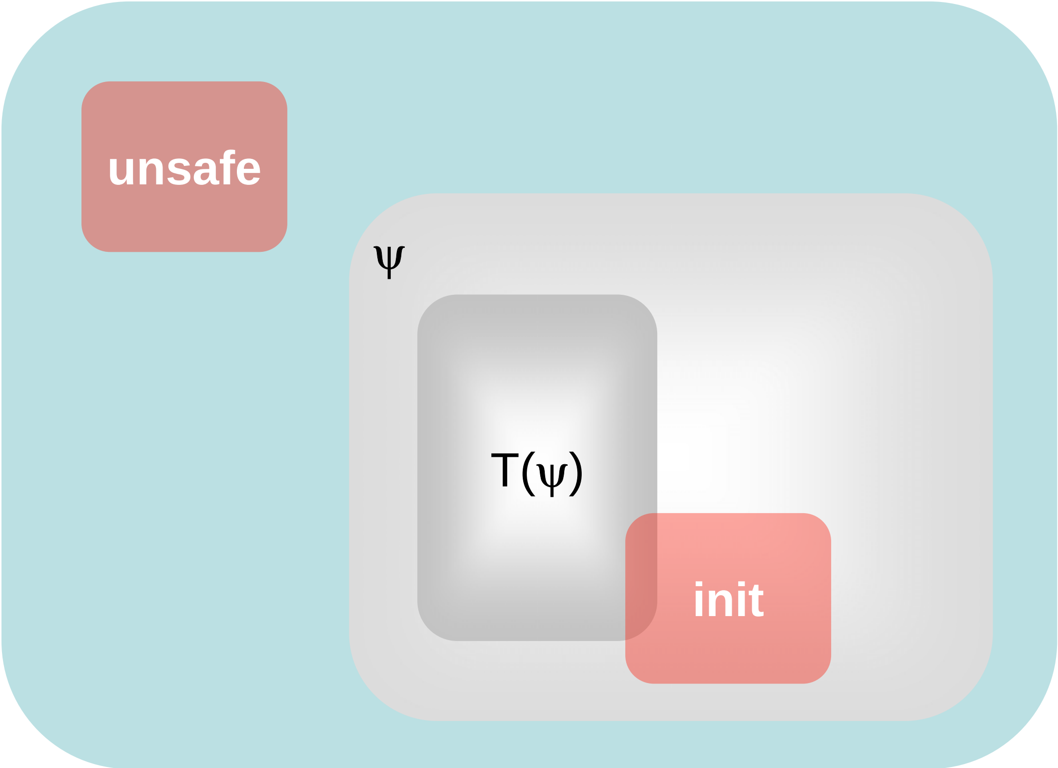
[Kurshan *et al*. '93]
[Clarke *et al*. '00]
[Ball, Rajamani '00]

Reachability

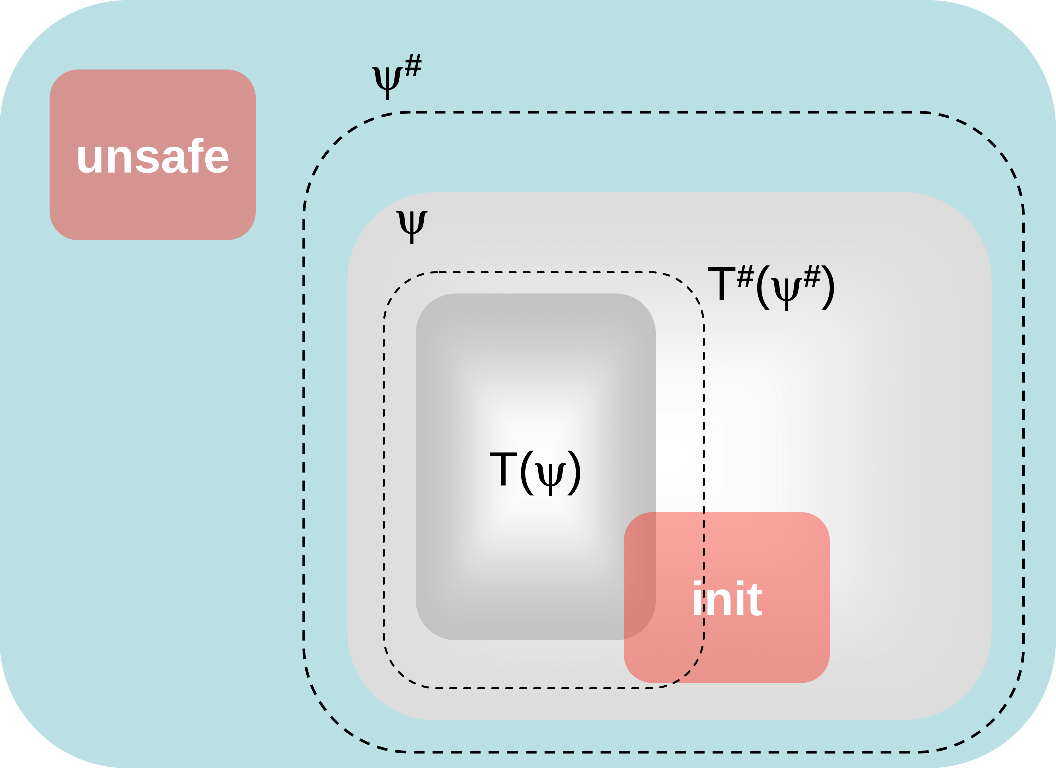


Safe Forward Invariants

- ψ is a safe forward invariant if
  - init $\Rightarrow \psi$
  - $T(\psi) \Rightarrow \psi$
  - $\psi \Rightarrow$ safe



Abstraction = Overapproximation of Behavior

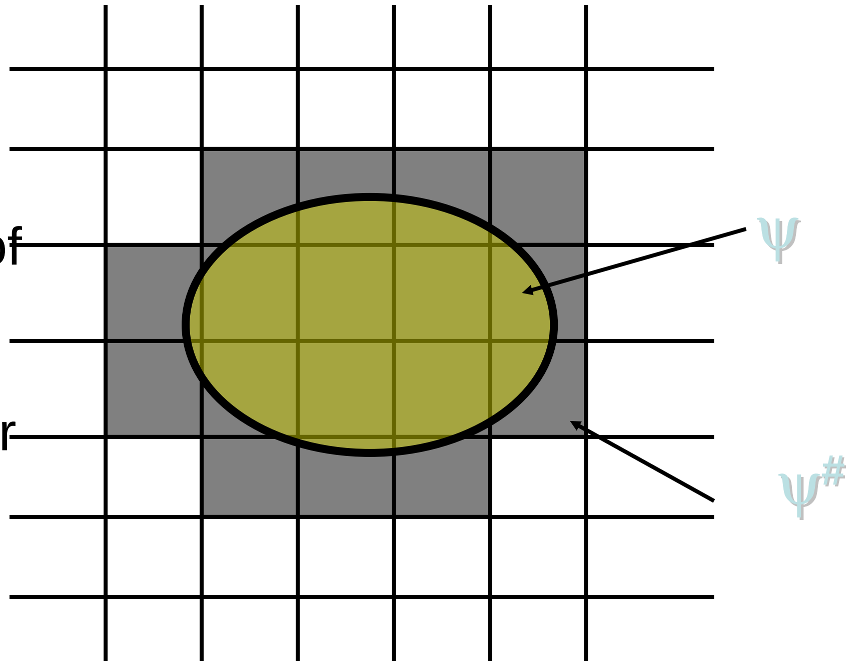


Predicate Abstraction for infinite-state systems

- Given set of predicates $F = \{ e_1, \ldots, e_k \}$
  - formulas describing properties of concrete system
- Create abstract system
  - set of abstract boolean variables $B = \{ b_1, \ldots, b_k \}$
    - b_i = true $\Leftrightarrow$ Set of states where e_i holds
- See
  - Abstract Interpretation, Cousot & Cousot ‘77
  - Graf & Saïdi, CAV ’97

Approximating concrete states

- Fundamental Operation
  - Approximating a set of concrete states by a set of predicates
  - Requires exponential number of theorem prover calls in worst case



Partitioning defined by the predicates

- Compute Symbolically
  - Main Operation

$$\exists X.\ [\psi \wedge (\wedge_i\, b_i \Leftrightarrow e_i\ )\ ]$$

Quantifier Elimination
Free Variables are *B variables*

Abstraction α and Concretization γ Functions

$\alpha : 2^C \longrightarrow A$ $\gamma : A \rightarrow 2^C$

Abstraction α and Concretization γ Functions

$\alpha : 2^C \longrightarrow A \qquad \gamma : A \rightarrow 2^C$

$2^C \cong \Psi \qquad A \cong \{0,1\}^k \cong \Psi^{\#}$

Abstraction α and Concretization γ Functions

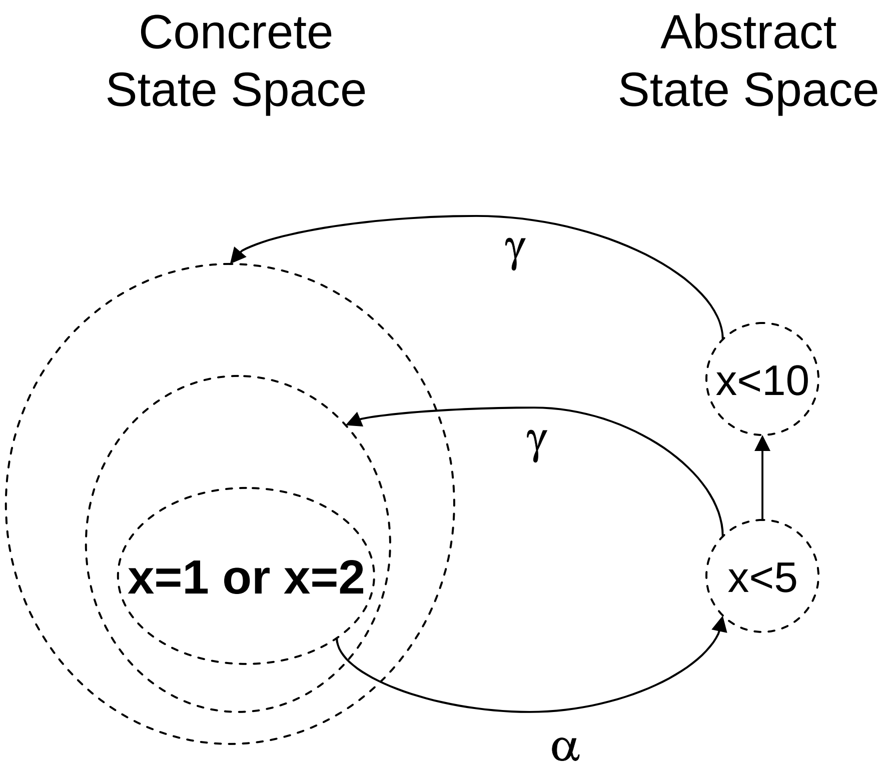
$\alpha : 2^C \longrightarrow A \qquad \gamma : A \rightarrow 2^C$

$2^C \approx \Psi \qquad A \approx 2^{2^k} \approx \Psi^{\#}$

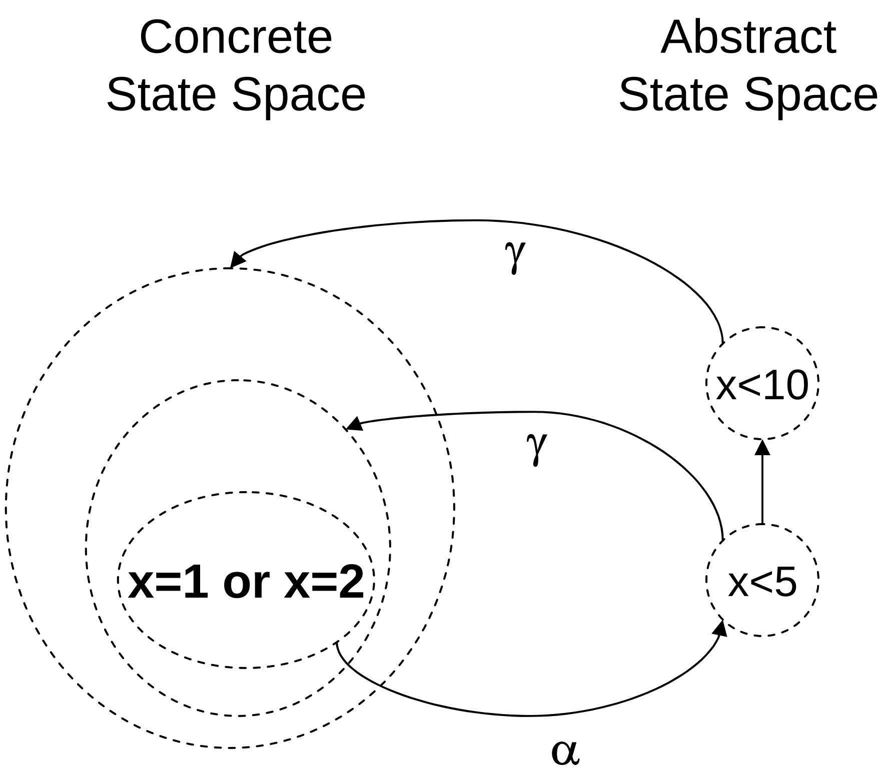
$\alpha(\psi) = \exists X.\ [\psi \wedge (\wedge_i\, b_i \Leftrightarrow e_i\,)\,]$

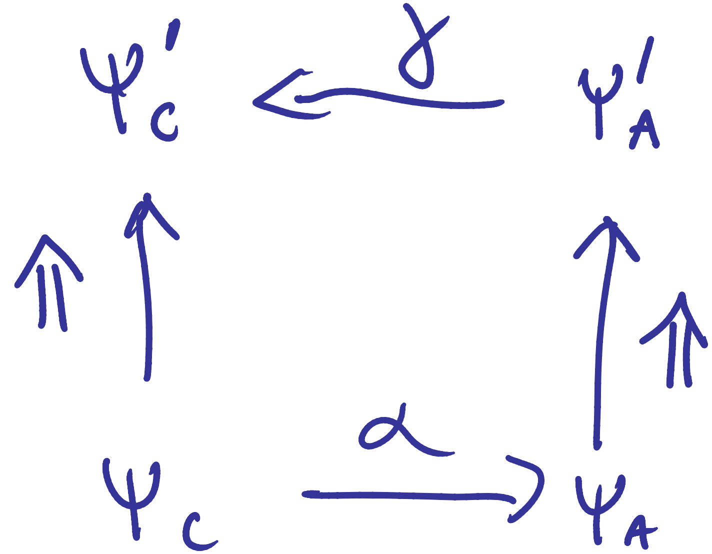
$\gamma(\psi^{\#}) = \psi^{\#}[b_1 \rightarrow e_1, \ldots b_k \rightarrow e_k]$

Abstraction = Overapproximation of Behavior

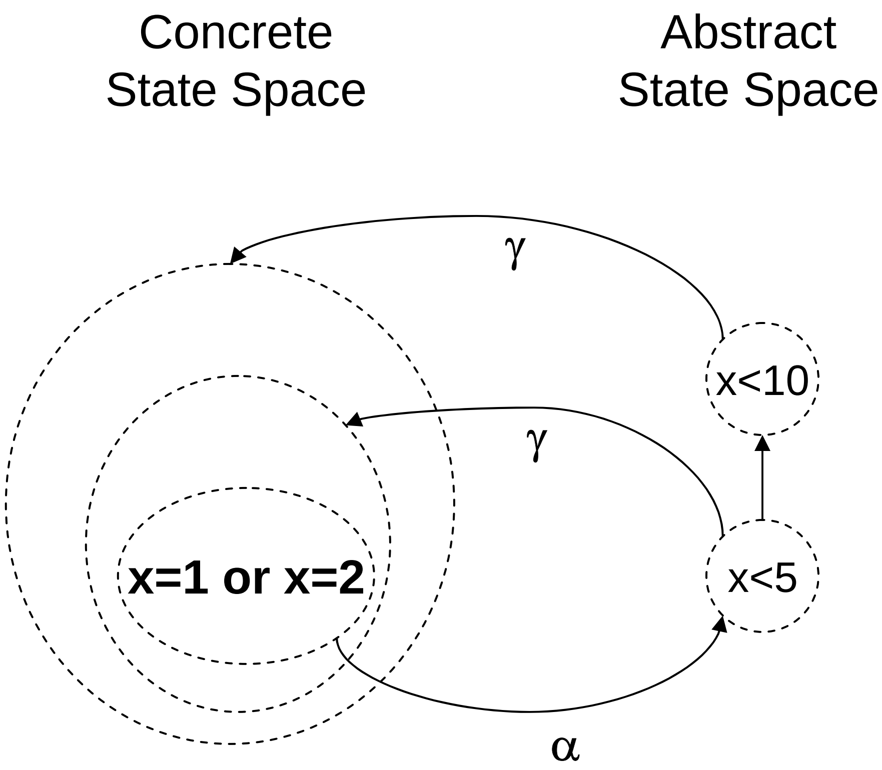


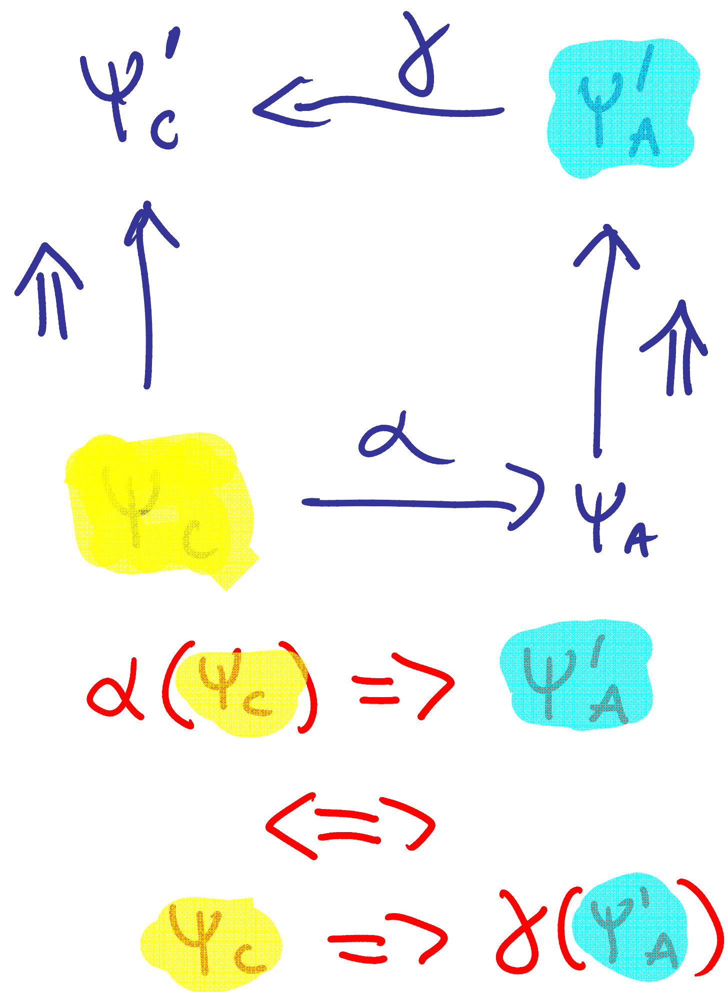
Abstraction = Overapproximation of Behavior





Abstraction =
Overapproximation of Behavior





Homework Example

$$\psi = (x = 1 \quad \vee \quad x = 2)$$

$$F = \{ e_1 : \; x < 5 \quad , \quad e_2 : \; x < 10 \}$$

$$B = \{ b_1 \qquad\qquad , \; b_2 \}$$

Abstract State Space

$(x < 5 \wedge x < 10)$ $b_1 \wedge b_2$

$(x \geq 5 \wedge x < 10)$ $\neg b_1 \wedge b_2$

$(x \geq 5 \wedge x \geq 10)$ $\neg b_1 \wedge \neg b_2$

$(x < 5 \wedge x \geq 10)$ $b_1 \wedge \neg b_2$

Boolean Covering

$$(x = 1 \lor x = 2) \Rightarrow (x < 5 \land x < 10) \qquad b_1 \land b_2$$

$$(x = 1 \lor x = 2) \Rightarrow (x \geq 5 \land x < 10) \qquad \neg b_1 \land b_2$$

$$(x = 1 \lor x = 2) \Rightarrow (x \geq 5 \land x \geq 10) \qquad \neg b_1 \land \neg b_2$$

Boolean Covering

$$(x < 5 \wedge x < 10) \quad b_1 \wedge b_2$$

$$(x = 1 \vee x = 2) \Rightarrow (x \geq 5 \wedge x < 10) \quad \neg b_1 \wedge b_2$$

$$(x \geq 5 \wedge x \geq 10) \quad \neg b_1 \wedge \neg b_2$$

Boolean Covering

$(x < 5 \land x < 10)$ $b_1 \land b_2$

$(x = 1 \lor x = 2)$

$$\alpha(\ (x = 1 \lor x = 2)\ ) = b_1 \land b_2$$

$$\gamma(\ b_1 \land b_2) = x < 5 \land x < 10$$

Instead

$$\alpha(\psi) = \exists X.\ [\psi \wedge (\wedge_i\, b_i \Leftrightarrow e_i\ )\ ]$$

$$\exists x.\ [(x=1 \vee x=2) \wedge (b_1 \Leftrightarrow x<5)$$

$$\wedge\ (b_2 \Leftrightarrow x<10)\ ]$$

Instead

$$\alpha(\psi) = \exists X.\ [\psi \wedge (\wedge_i\, \boldsymbol{b}_i \Leftrightarrow \boldsymbol{e}_i\ )\ ]$$

$$\exists x.\ [(x=1 \vee x=2) \wedge (b_1 \Leftrightarrow x<5)$$
$$\wedge\ (b_2 \Leftrightarrow x<10)\ ]$$

$$\swarrow\ x=1$$

$$(T) \wedge (b_1 \Leftrightarrow 1<5)$$
$$\wedge\ (b_2 \Leftrightarrow 1<10)$$

Instead

$$\alpha(\psi) = \exists X.\ [\psi \wedge (\wedge_i\, b_i \Leftrightarrow e_i\ )\ ]$$

$$\exists x.\ [(x=1 \vee x=2) \wedge (b_1 \Leftrightarrow x<5)$$
$$\wedge\ (b_2 \Leftrightarrow x<10)\ ]$$

$\swarrow$ $x=1$

$$b_1 \wedge b_2$$

Instead

$$\alpha(\psi) = \exists X.\ [\psi \wedge (\wedge_i\, b_i \Leftrightarrow e_i\ )\ ]$$

$$\exists x.\ [(x=1 \vee x=2) \wedge (b_1 \Leftrightarrow x<5) \wedge (b_2 \Leftrightarrow x<10)]$$

$x=1$:

$$b_1 \wedge b_2$$

$x=2$:

$$(T) \wedge (b_1 \Leftrightarrow 2<5) \wedge (b_2 \Leftrightarrow 2<10)$$

Instead

$$\alpha(\psi) = \exists X.\ [\psi \wedge (\wedge_i\, b_i \Leftrightarrow e_i\ )\ ]$$

$$\exists x.\ [(x=1 \vee x=2) \wedge (b_1 \Leftrightarrow x<5)$$
$$\wedge\ (b_2 \Leftrightarrow x<10)\ ]$$

$x=1$ ↙ ↘ $x=2$

$$b_1 \wedge b_2 \quad \vee \quad b_1 \wedge b_2$$

Instead

$$\alpha(\psi) = \exists X.\ [\psi \wedge (\wedge_i\, b_i \Leftrightarrow e_i\ )\ ]$$

$$\exists x.\ [(x=1 \vee x=2) \wedge (b_1 \Leftrightarrow x < 5)$$
$$\wedge\ (b_2 \Leftrightarrow x < 10)\ ]$$

$$=$$

$$b_1 \wedge b_2$$

Counterexample-driven Refinement

```
F := {};
loop
    T# := predAbs(T,F)
    if unsafe ∉ lfp(T#, init) then
        return SUCCESS
    else
        find min k s.t. unsafe ∈ T#^k(init)
        if unsafe ∈ T^k(init) then
            return FAILURE
        else
            find G s.t. unsafe ∉ U#^k(init)
                where U# = predAbs(T,G)
            F := F ∪ G
forever
```

C-

Types	**τ**	::=	void | bool | int | ref τ
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$ | &x | *x
LExpression	**l**	::=	x | *x
Declaration	**d**	::=	τ $x_1, x_2, \ldots, x_n$
Statements	**s**	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	**l** = **e**
		|	**l** = f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
		|	return x
		|	$\mathbf{s_1}$; $\mathbf{s_2}$;… ; $\mathbf{s_n}$
Procedures	**p**	::=	τ f (x_1: τ_1, x_2 : τ_2, …, x_n : τ_n)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ … $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ … $\mathbf{p_n}$

C--

Types	τ	::=	void | bool | int
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$
LExpression	**l**	::=	x
Declaration	**d**	::=	τ $x_1, x_2, \ldots, x_n$
Statements	**s**	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	**l** = **e**
		|	f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
		|	return
		|	$\mathbf{s_1}$; $\mathbf{s_2}$;… ; $\mathbf{s_n}$
Procedures	**p**	::=	f (x_1: τ_1, x_2 : τ_2, …, x_n : τ_n)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ … $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ … $\mathbf{p_n}$

BP

Types	τ	::=	void | bool
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$
LExpression	**l**	::=	x
Declaration	**d**	::=	τ $x_1, x_2, \ldots, x_n$
Statements	**s**	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	**l** = **e**
		|	f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
		|	return
		|	$\mathbf{s_1}$; $\mathbf{s_2}$;… ; $\mathbf{s_n}$
Procedures	**p**	::=	f ($x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ … $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ … $\mathbf{p_n}$

What is Hard?

- Abstracting
  - from a language with pointers (C)
  - to one without pointers (boolean programs)

- All side effects need to be modeled by copying (as in dataflow)

Syntactic sugar

```
if (e) {
    S1;
} else {
    S2;
}
S3;
```

```
goto L1, L2;

    L1:  assume(e);
    S1;
    goto L3;

    L2: assume(!e);
    S2;
    goto L3;

L3: S3;
```

Example, in C

```
int g;

main(int x, int y){

  cmp(x, y);

  if (!g) {
    if (x != y)
      assert(0);
  }
}
```

```
void cmp (int a , int b) {
  if (a == b)
        g = 0;
  else
        g = 1;
}
```

Example, in C--

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp(int a , int b) {
   goto L1, L2;

   L1: assume(a==b);
       g = 0;
       return;

   L2: assume(a!=b);
       g = 1;
       return;
}
```

c2bp: Predicate Abstraction for C Programs

Given

- P : a C program
- F = {e_1,...,e_n}
  - each e_i a pure boolean expression
  - each e_i represents set of states for which e_i is true

Produce a *boolean program* B(P,F)

- same control-flow structure as P
- boolean vars {b_1,...,b_n} to match {e_1,...,e_n}
- properties true of B(P,F) are true of P

Assumptions

Given

- P : a C program
- F = {e_1,...,e_n}
  - each e_i a pure boolean expression
  - each e_i represents set of states for which e_i is true

- Assume: each e_i uses either:
  - only globals (global predicate)
  - local variables from some procedure (local predicate for that procedure)

- Mixed predicates:
  - predicates using both local variables and global variables
  - complicate “return” processing
  - covered in advanced topics

C2bp Algorithm

- Performs modular abstraction
  - abstracts each procedure in isolation
- Within each procedure, abstracts each statement in isolation
  - no control-flow analysis
  - no need for loop invariants

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
   goto L1, L2

   L1: assume(a==b);
         g = 0;
         return;

  L2: assume(a!=b);
         g = 1;
         return;
}
```

Preds: {x==y}

{g==0}

{a==b}

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
   goto L1, L2

   L1: assume(a==b);
         g = 0;
         return;

  L2: assume(a!=b);
         g = 1;
         return;
}
```

```
decl {g==0} ;

main( {x==y} ) {

}
```

Preds: {x==y}

{g==0}

{a==b}

```
void cmp ( {a==b} ) {

}
```

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
  goto L1, L2

  L1: assume(a==b);
        g = 0;
        return;

  L2: assume(a!=b);
        g = 1;
        return;
}
```

```
decl {g==0} ;

main( {x==y} ) {

  cmp( {x==y} );

  assume( {g==0} );
  assume( !{x==y} );
  assert(0);
}
```

Preds: {x==y}

{g==0}

{a==b}

```
void cmp ( {a==b} ) {
  goto L1, L2;

  L1: assume( {a==b} );
        {g==0} = T;
        return;

  L2: assume( !{a==b} );
        {g==0} = F;
        return;
}
```

C--

Types	τ	::=	void | bool | int
Expressions	$\mathbf{e}$	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$
LExpression	$\mathbf{l}$	::=	x
Declaration	$\mathbf{d}$	::=	τ $x_1, x_2, \ldots, x_n$
Statements	$\mathbf{s}$	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
			| assume($\mathbf{e}$)
			| $\mathbf{l}$ = $\mathbf{e}$
			| f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
			| return
			| $\mathbf{s_1}$; $\mathbf{s_2}$;… ; $\mathbf{s_n}$
Procedures	$\mathbf{p}$	::=	f ($x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$)
Program	$\mathbf{g}$	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ … $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ … $\mathbf{p_n}$

Abstracting Assigns via WP

- Statement y=y+1 and F={ y<4, y<5 }
  - {y<4}, {y<5} = ((!{y<5} || !{y<4}) ? F : *), {y<4} ;
- WP(x=e,Q) = Q[x -> e]
- WP(y=y+1, y<5) =

  (y<5) [y -> y+1] =

  (y+1<5) =

  (y<4)

WP Problem

- WP(s, e_i) not always expressible via { $e_1,\ldots,e_n$ }
- Example
  - F = { x==0, x==1, x<5 }
  - WP(x=x+1 , x<5) = x<4
  - Best possible: x==0 || x==1

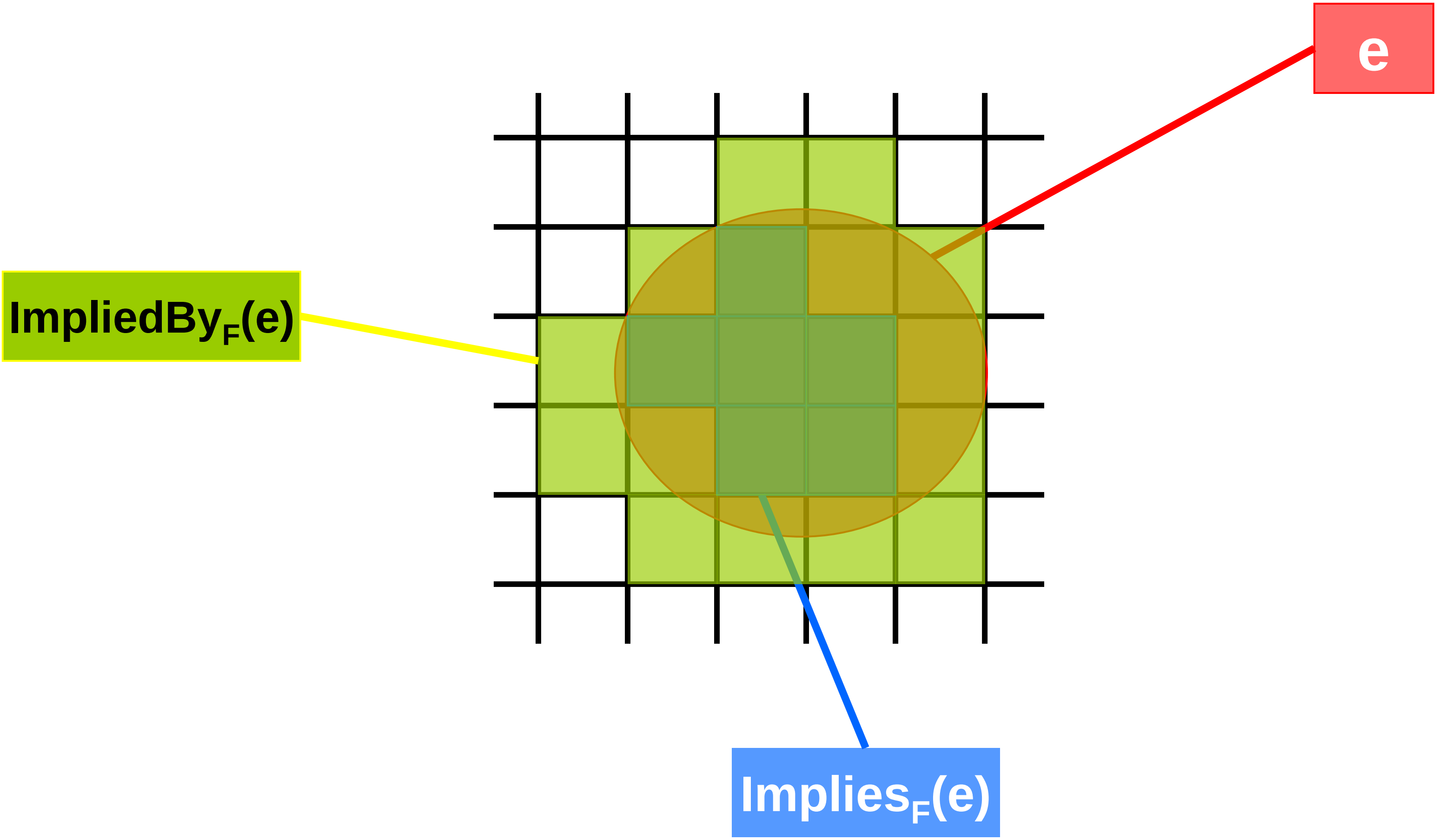
Abstracting Expressions via F

- $F = \{ e_1, \ldots, e_n \}$

- $Implies_F(e)$
  - *best* boolean function over F *that implies* e

- $ImpliedBy_F(e)$
  - *best* boolean function over F *implied by* e
  - $ImpliedBy_F(e) = !Implies_F(!e)$

$\text{Implies}_F(e)$ and $\text{ImpliedBy}_F(e)$



Computing $\text{Implies}_F(e)$

- *minterm* $m = d_1 \ \&\& \ \ldots \ \&\& \ d_n$
  - where $d_i = e_i$ or $d_i = !e_i$

- $\text{Implies}_F(e)$
  - disjunction of all minterms that imply e

- Naïve approach
  - generate all 2^n possible minterms
  - for each minterm m, use decision procedure to check *validity* of each implication $m \Rightarrow e$

- Many optimizations possible

Abstracting Assignments

- if $Implies_F(WP(s, e_i))$ is true before s then
  - e_i is true after s
- if $Implies_F(WP(s, !e_i))$ is true before s then
  - e_i is false after s

$$\{e_i\} = \begin{array}{lll} Implies_F(WP(s, e_i)) & ? & true : \\ Implies_F(WP(s, !e_i)) & ? & false \\ & : & *; \end{array}$$

Assignment Example

Statement in P:

y = y+1;

Predicates in F:

{x==y}

Weakest Precondition:

WP(y=y+1, x==y) = x==y+1

Implies_F(x==y+1) = ?

Implies_F(x!=y+1) = ?

Assignment Example

Statement in P:

y = y+1;

Predicates in F:

{x==y}

Weakest Precondition:

WP(y=y+1, x==y) = x==y+1

Implies_F(x==y+1) = false

Implies_F(x!=y+1) = x==y

Abstraction of assignment in B:

{x==y} = {x==y} ? false : *;

Abstracting Assumes

- WP(assume(e) , Q) = $e \Rightarrow Q$

- assume(e) is abstracted to:

  assume(ImpliedBy_F(e))

- Example:

  F = {x==2, x<5}

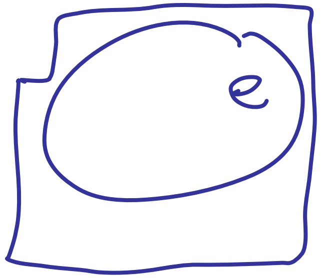
  assume(x < 2) is abstracted to:

  assume({x<5} && !{x==2})

Assume, Explained

if assume evaluates true in C program then it must evaluate true in boolean program

C	BP
assume(e)	assume(ImpliedBy(e))



Abstracting Procedures

- Each predicate in F is annotated as being either global or local to a particular procedure

- Procedures abstracted in two passes:
  – a *signature* is produced for each procedure in isolation
  – procedure calls are abstracted given the callees' signatures

Abstracting a procedure call

- Procedure call
  - a sequence of assignments from actuals to formals
  - see assignment abstraction
- Procedure return
  - NOP for C-- with assumption that all predicates mention either only globals or only locals
  - with pointers and with mixed predicates:
    - Most complicated part of c2bp
    - Covered in the advanced topics section

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a==b);
         g = 0;
         return;

  L2: assume(a!=b);
         g = 1;
         return;
}
```

```
decl {g==0} ;

main( {x==y} ) {

  cmp( {x==y} );

  assume( {g==0} );
  assume( !{x==y} );
  assert(0);
}
```

{x==y}

{g==0}

{a==b}

```
void cmp ( {a==b} ) {
Goto L1, L2

   L1: assume( {a==b} );
         {g==0} = T;
         return;

  L2: assume( !{a==b} );
         {g==0} = F;
         return;
}
```

Precision

- For program P and F = $\{e_1,...,e_n\}$, there exist two “ideal” abstractions:
  - Boolean(P,F) : most precise abstraction
  - Cartesian(P,F) : less precise abstraction, where each boolean variable is updated independently
  - [See Ball-Podelski-Rajamani, TACAS 00]

- Theory:
  - with an “ideal” theorem prover, c2bp can compute Cartesian(P,F)

- Practice:
  - c2bp computes a less precise abstraction than Cartesian(P,F)
  - we use Das/Dill’s technique to incrementally improve precision
  - with an “ideal” theorem prover, the combination of c2bp + Das/Dill can compute Boolean(P,F)

C-

Types	τ	::=	void | bool | int | ref τ
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$ | &x | *x
LExpression	**l**	::=	x | *x
Declaration	**d**	::=	τ $x_1, x_2, \ldots, x_n$
Statements	**s**	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	**l** = **e**
		|	**l** = f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
		|	return x
		|	$\mathbf{s_1}$; $\mathbf{s_2}$;… ; $\mathbf{s_n}$
Procedures	**p**	::=	τ f (x_1: τ_1, x_2 : τ_2, …, x_n : τ_n)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ … $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ … $\mathbf{p_n}$

Pointers and SLAM

- With pointers, C supports call by reference
  - Strictly speaking, C supports only call by value
  - With pointers and the address-of operator, one can simulate call-by-reference
- Boolean programs support only call-by-value-result
  - SLAM mimics call-by-reference with call-by-value-result
- Extra complications:
  - address operator (&) in C
  - multiple levels of pointer dereference in C

Assignments + Pointers

Statement in P:

*p = 3

Predicates in E:

{x==5}

Weakest Precondition:

WP(*p=3 , x==5) = x==5

What if *p and x alias?

Correct Weakest Precondition:

(p==&x and 3==5) or (p!=&x and x==5)

We use Das's pointer analysis [PLDI 2000] to prune disjuncts representing infeasible alias scenarios.

Abstracting Procedure Return

- Need to account for
  - lhs of procedure call
  - mixed predicates
  - side-effects of procedure

- Boolean programs support only call-by-value-result
  - C2bp models all side-effects using return processing

Abstracting Procedure Returns

- Let a be an actual at call-site P(…)
  - pre(a) = the value of a before transition to P

- Let f be a formal of a procedure P
  - pre(f) = the value of f upon entry to P

predicate | call/return relation | call/return assign

```
Q() {
  int x = 1;        {x==1}
  x = R(x)          {x==2}
}
```

f = x

pre(f) == pre(x)

x = r

```
int R (int f) {
  int r = f+1;      {f==pre(f)}
  f = 0;            {r==pre(f)+1}
  return r;
}
```

WP(f=x, f==pre(f)) = x==pre(f)

x==pre(f) is true at the call to R

WP(x=r, x==2) = r==2

pre(f)==pre(x) **and** pre(x)==1 **and** r==pre(f)+1 **implies** r==2

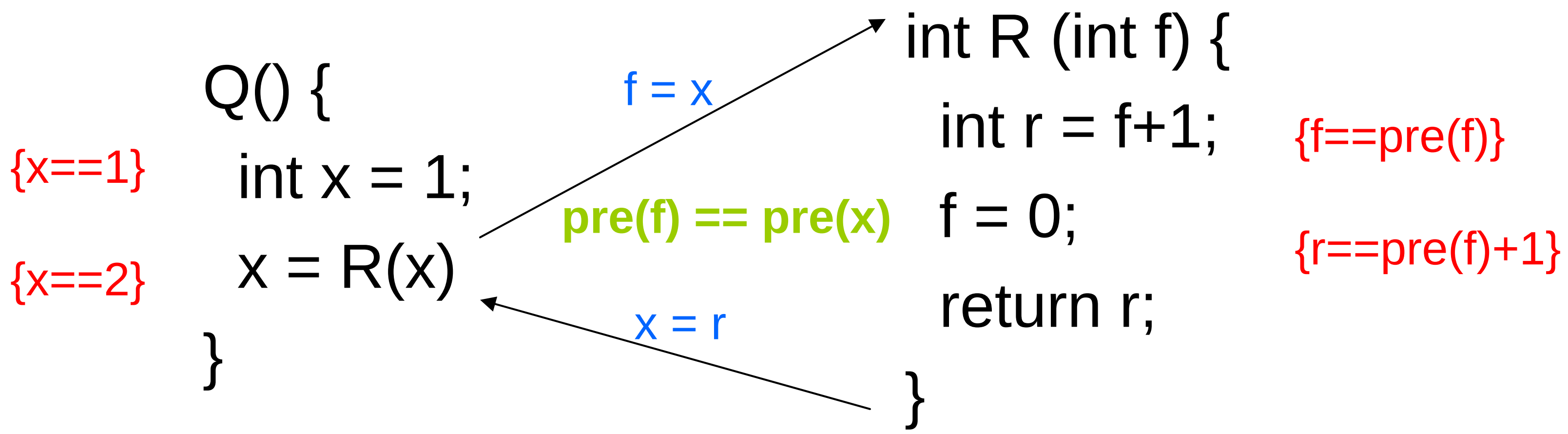
{x==1}

s

```
Q() {
 {x==1},{x==2} = T,F;
 s = R(T);
 {x==2} = s & {x==1};
}
```

```
bool R ( {f==pre(f)} ) {
  {r==pre(f)+1} = {f==pre(f)};
  {f==pre(f)} = *;
  return {r==pre(f)+1};
}
```

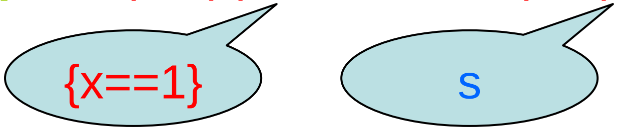
predicate | call/return relation | call/return assign



WP(f=x, f==pre(f)) = x==pre(f)

x==pre(f) is true at the call to R

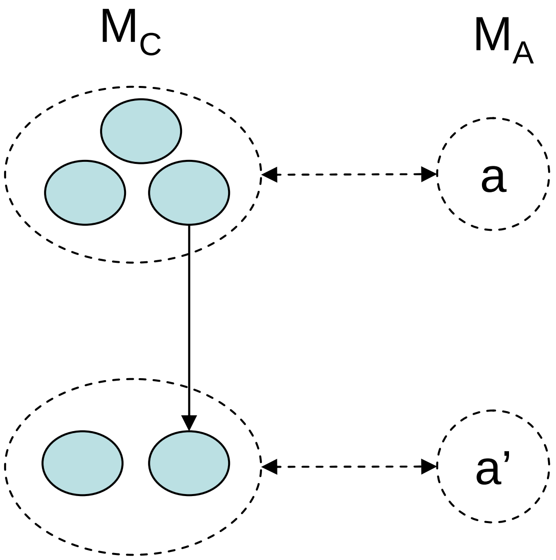
WP(x=r, x==2) = r==2 pre(f)==pre(x) **and** pre(x)==1 **and** r==pre(f)+1 **implies** r==2



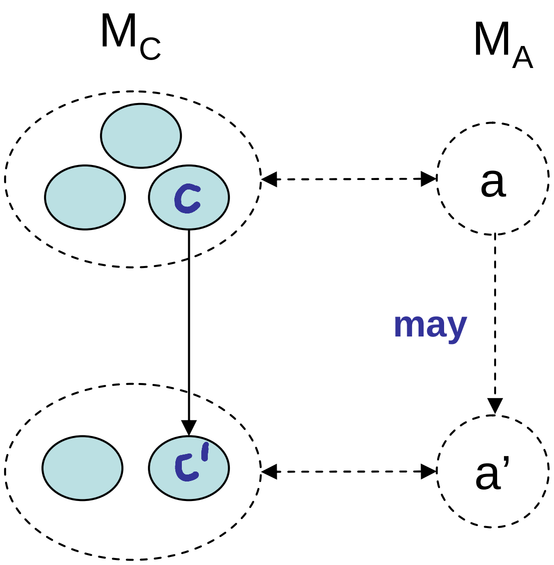

```
Q() {
 {x==1},{x==2} = T,F;

 s = R(T);

 {x==1}, {x==2} =  *, s & {x==1};
}
```

```
bool R ( {f==pre(f)} ) {
  {r==pre(f)+1} = {f==pre(f)};
  {f==pre(f)} = *;
  return {r==pre(f)+1};
}
```

Last Word on Abstraction



Last Word on Abstraction



$$a \rightarrow a' \text{ if}$$

$$\exists c\, \exists c' : c \sim a \wedge c' \sim a' \wedge c \rightarrow c'$$

predicate | call/return relation | call/return assign

```
Q() {
  int x = 1;
  x = R(x)
}
```

{x==1}

{x==2}

f = x

pre(f) == pre(x)

x = r

```
int R (int f) {
  int r = f+1;
  f = 0;
  return r;
}
```

{f==pre(f)}

{r==pre(f)+1}

predicate | call/return relation | call/return assign

```
Q() {
  int x = 1;        {x==1}
  x = R(x)          {x==2}
}
```

f = x

pre(f) == pre(x)

x = r

```
int R (int f) {
  int r = f+1;      {f==pre(f)}
  f = 0;            {r==pre(f)+1}
  return r;
}
```

$$WP(x=r,\ x==2) = (r==2)$$

$$F_Q = \{ x==1,\ x==2 \}$$

$$\text{Implies}_{F_Q}(r==2) = \text{false}$$

predicate | call/return relation | call/return assign

```
Q() {
  int x = 1;      {x==1}
  x = R(x)        {x==2}
}

int R (int f) {
  int r = f+1;    {f==pre(f)}
  f = 0;          {r==pre(f)+1}
  return r;
}
```

f = x

pre(f) == pre(x)

x = r

$$WP(x=r,\ x==2) = (r==2)$$

$$F'_Q = \{pre(x)==1,\ pre(x)==2\} \cup \{f==pre(f),\ r==pre(f)+1\} \cup \{pre(f)==pre(x)\}$$

$$WP(x=r,\ x==2) = (r==2)$$

$$F_Q' = \{\ pre(x)==1,\ \ pre(x)==2\ \}$$
$$\cup\ \{\ f==pre(f),\ r==pre(f)+1\ \}$$
$$\cup\ \{\ pre(x)==pre(f)\ \}$$

$$Implies_{F_Q'}(r==2) =$$

$$pre(x)==1 \wedge pre(x)==pre(f) \wedge r=pre(f)+1$$

Extending Pre-states

- Suppose formal parameter is a pointer
  - eg. P(int *f)

- pre(*f)
  - value of *f upon entry to P
  - can't change during P

- * pre(f)
  - value of dereference of pre(f)
  - can change during P

predicate | call/return relation | call/return assign

{a==pre(a)}
{*pre(a)==pre(*a)}

```
Q() {
int x = 1;
R(&x);
}
```

{x==1}

{x==2}

a = &x

&x = pre(a)
pre(x) = pre(*a)

```
int R (int *a) {
*a = *a+1;
}
```

{*pre(a)=pre(*a)+1}

pre(x)==1 **and** pre(x)==pre(*a) **and** *pre(a)==pre(*a)+1 **and** pre(a)==&x
implies x==2

{x==1}

s

```
Q() {
 {x==1},{x==2} = T,F;
 s = R(T,T);
 {x==2} = s & {x==1};
}
```

```
bool R ( {a==pre(a)}, {*pre(a)==pre(*a)} ) {
 {*pre(a)==pre(*a)+1} = {*pre(a)==pre(*a)} ;
return {*pre(a)==pre(*a)+1};
}
```

For Fun

- Show that call-by-reference can be *simulated* by call-by-value result

- or

- Create a reachability algorithm for boolean programs with references
  – T := void | bool | ref T;

Counterexample-driven Refinement

F := **{};**
loop
    $T^{\#}$:= predAbs(T,F)
    if unsafe $\notin$ lfp($T^{\#}$, init) **then**
        return SUCCESS
    else
        find min k s.t. unsafe $\in$ $T^{\# k}$ (init)
        if unsafe $\in$ T^k (init) **then**
            return FAILURE
        else
            find G s.t. unsafe $\notin$ $U^{\# k}$ (init)
                where $U^{\#}$ = predAbs(T,G)
            F := F $\cup$ G
forever

Example

```
bool id (bool b) {
    decl r;
 L1: r := !b;
 L2: r := !r;
 L3: return r;
}
```

Example

```
bool id (bool b) {
    decl r;
                    ──> (L1, b→1)
L1: r := !b;
L2: r := !r;
L3: return r;
}
```

Example

```
bool id (bool b) {
    decl r;
 L1: r := !b;
 L2: r := !r;
 L3: return r;
}
```

(L1, b → 1)

(L2, b → 1, r → 0)

Example

```
bool id (bool b) {
    decl r;
 L1: r := !b;
 L2: r := !r;
 L3: return r;
}
```

→ (L1, b → 1)

→ (L2, b → 1, r → 0)

→ (L3, b → 1, r → 1)

Example

```
bool id(bool b) {
    decl r;
L1: r := !b;
L2: r := !r;
L3: return r;
}
```

($L1, b \to 1$)
($L1, b \to 0$)

($L2, b \to 1, r \to 0$)
($L2, b \to 0, r \to 1$)

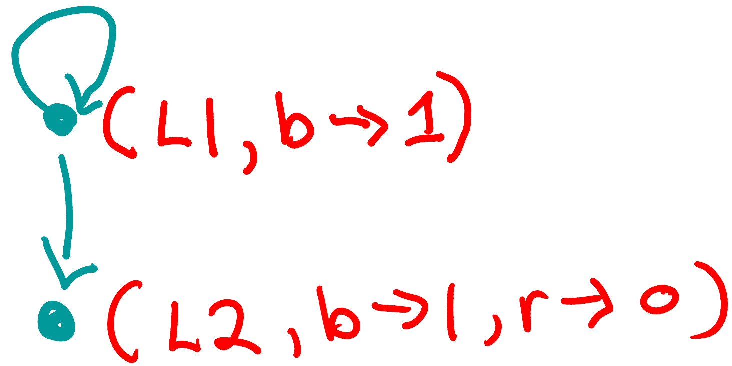
($L3, b \to 1, r \to 1$)
($L3, b \to 0, r \to 0$)

Example

```
bool id(bool b) {
    decl r;
 L1: r := !b;
 L2: r := !r;
 L3: return r;
}
```

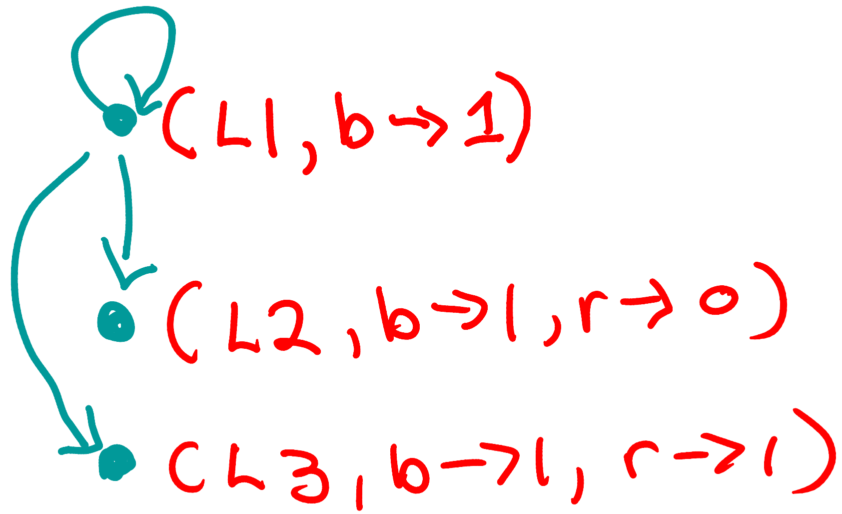
Example

```
bool id (bool b) {
    decl r;
  L1: r := !b;
  L2: r := !r;
  L3: return r;
}
```



Example

```
bool id (bool b) {
    decl r;
  L1: r := !b;
  L2: r := !r;
  L3: return r;
}
```



Reachability in Boolean Programs

- Algorithm based on CFL reachability
  - [Sharir-Pnueli 81] [Reps-Sagiv-Horwitz 95]

- “path edge” of procedure P
  - <entry,d1> $\rightarrow$ <s2,d2>
  - “if P’s entry is reachable in state d1 then statement s2 of P is reachable in state d2”

- “summary edge” of procedure P
  - <call Q,d1> $\rightarrow$ <ret,d2>
  - “if P calls Q from state d1 then Q returns to P in state d2”

Symbolic CFL reachability

- Partition path edges by their “target”
  - PE(s) = { <d1,d2> | <entry,d1> → <s,d2> }

- What is <d1,d2> for boolean programs?
  - A bit-vector!

- What is PE(s)?
  - A set of bit-vectors

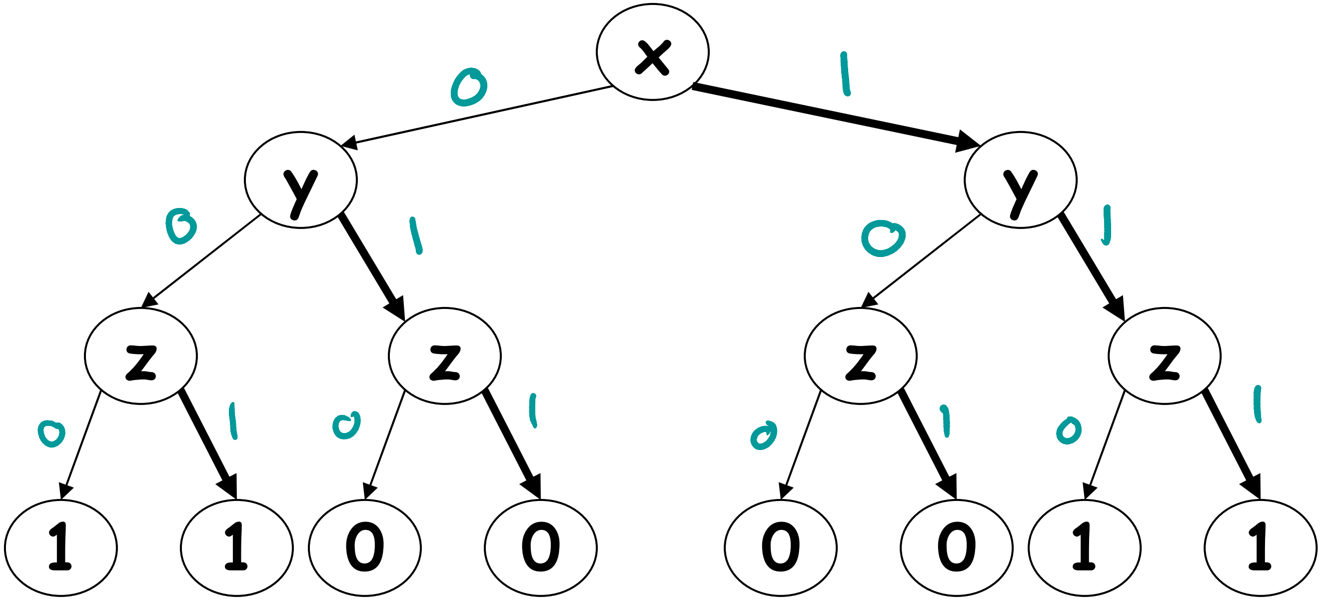
- Use a BDD (attached to s) to represent PE(s)

Binary Decision Diagrams

- Acyclic graph data structure for representing a boolean function (equivalently, a set of bit vectors)
- $F(x,y,z) = (x=y)$

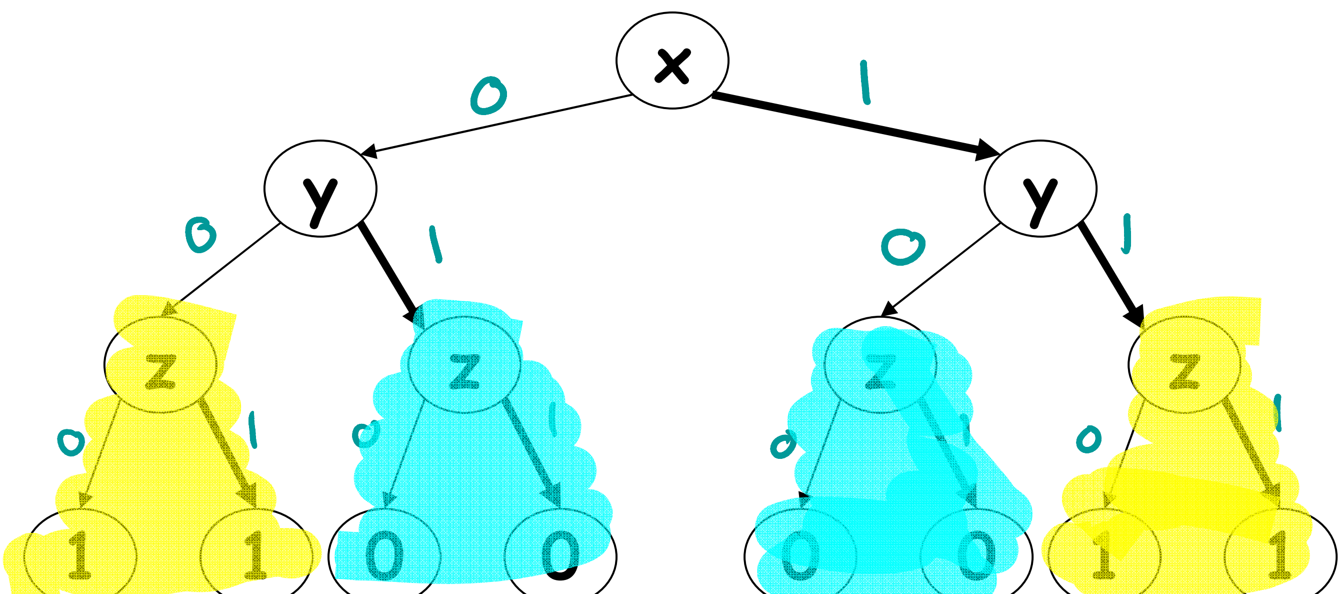
Binary Decision Diagrams

- Acyclic graph data structure for representing a boolean function (equivalently, a set of bit vectors)
- F(x,y,z) = (x=y)

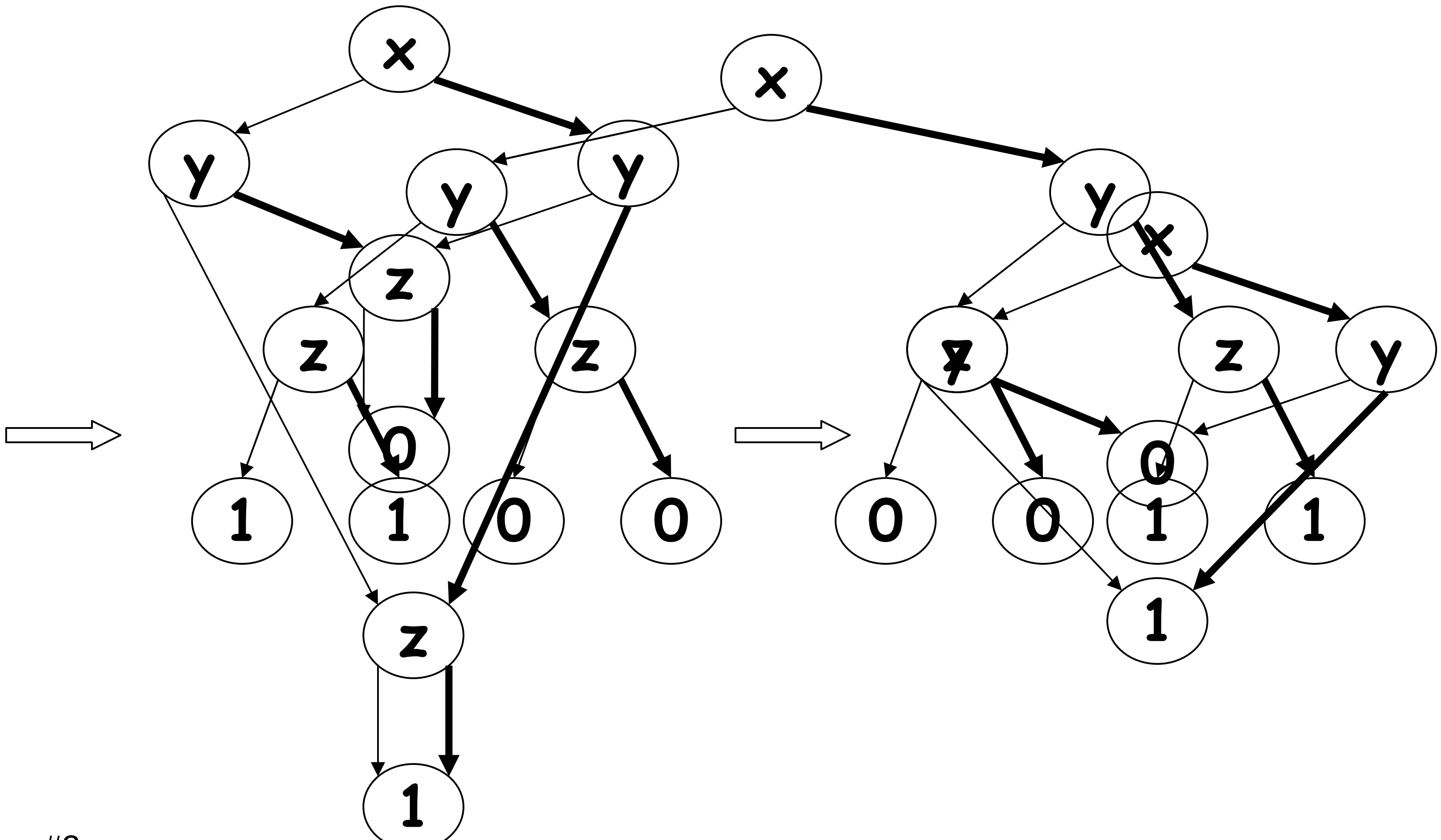


Binary Decision Diagrams

- Acyclic graph data structure for representing a boolean function (equivalently, a set of bit vectors)
- F(x,y,z) = (x=y)



Hash Consing + Variable Elimination



Binary Decision Diagrams

- Canonical representation of
  - boolean functions
  - set of (fixed-length) bitvectors
  - binary relations over finite domains

- Efficient algorithms for common operations
  - transfer function
  - join/meet
  - subsumption test

```
void cmp ( e2 ) {
[5]Goto L1, L2
   [6]L1: assume( e2 );
   [7] gz = T; goto L3;

   [8]L2: assume( !e2 );
   [9]gz = F; goto L3

[10] L3: return;
}
```

BDD at line [10] of cmp:

e2=e2′ & gz′=e2′

Read: “cmp leaves e2 unchanged and sets gz to have the same final value as e2”

More Detail

`e2=e2′ & gz′=e2′`

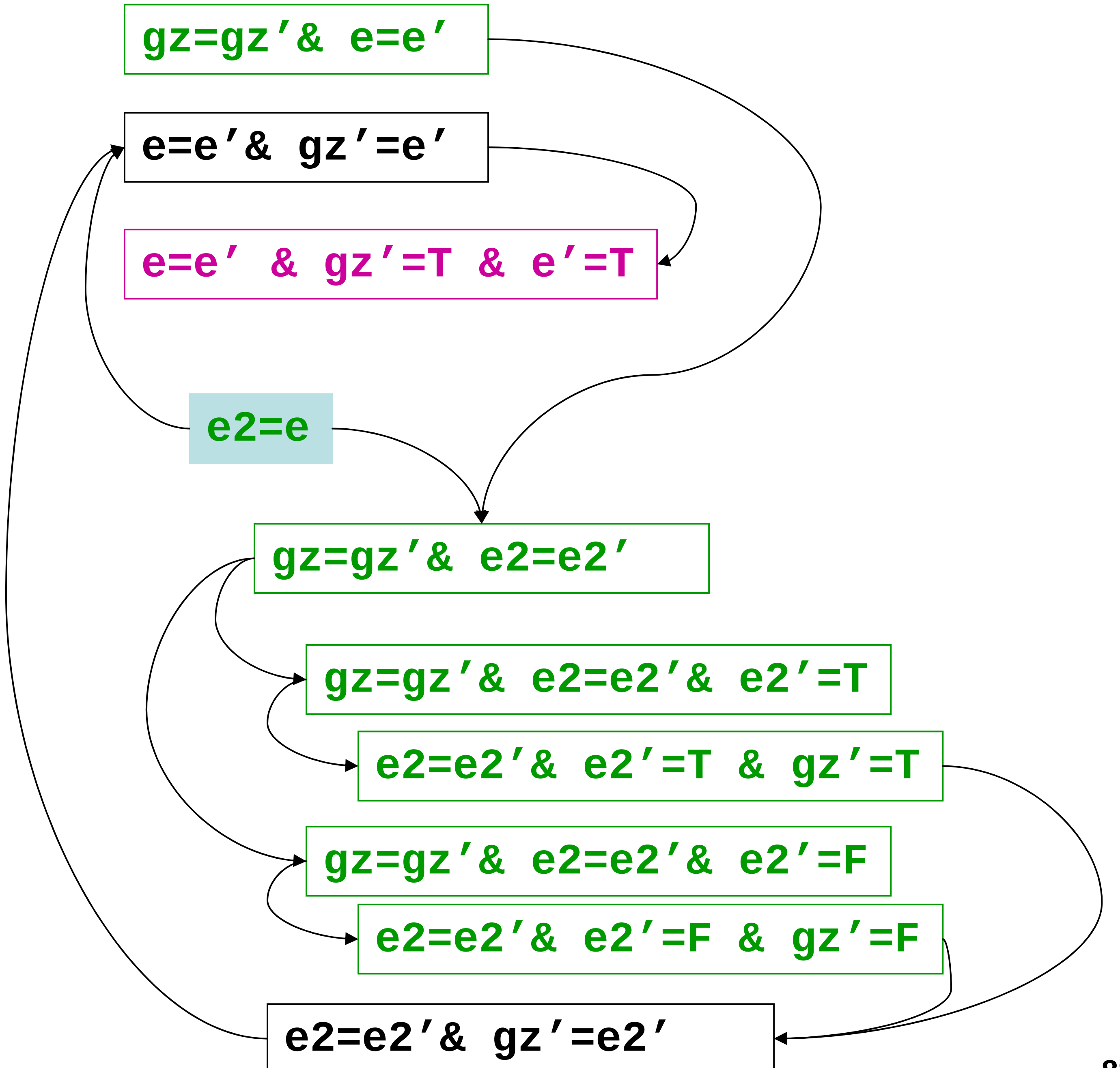
(entry, e2→T, gz→F) → (exit, e2→T, gz→T)

(entry, e2→T, gz→T) → (exit, e2→T, gz→T)

(entry, e2→F, gz→F) → (exit, e2→F, gz→F)

(entry, e2→F, gz→T) → (exit, e2→F, gz→F)

```
decl gz ;
main( e ) {

 [1] equal( e );

 [2] assume( gz );

 [3] assume( !e );

 [4] assert(F);
}

void cmp ( e2 ) {
[5]Goto L1, L2

  [6]L1: assume( e2 );

  [7] gz = T; goto L3;

  [8]L2: assume( !e2 );

  [9]gz = F; goto L3

[10] L3: return;
}
```



Reachability Summary

- Explicit representation of CFG
- Implicit representation of path edges and summary edges
- Generation of hierarchical error traces

- Complexity: $O(E * 2^{O(N)})$
  - E is the size of the CFG
  - N is the max. number of variables in scope