Over- and Underapproximations in Program Analysis

A Possible Problem

```
f(a) {
    x = unknown(…);
     if (x == 0) …
}
```

- Value of x is unknown
  - but predicates involving x will accumulate

Does This Happen Very Often?

- Yes!
- Unknowable predicates arise from
  - Unknown functions (e.g., no code available)
  - I/O functions
  - Non-deterministic functions
    - E.g., malloc
  - Anything that is too hard to analyze

Example

```
void foo(int a) {
   int *t = malloc(…);
   char u = getUserInput();
   if ((t != NULL) || (u ==  '\n' )) && (a == FLAG)
         return 0;
   else return 1;
}
```

Condition under which 0 is returned:

`((t != NULL) || (u ==  '\n' )) && (a == FLAG)`

But from a caller of foo, only a == FLAG is useful

The Plan

- Define a small language
- Show how to compute predicates
- Show how to simplify predicates
  - Interprocedurally
  - With recursive functions

A Small Language

- Functions
  - $f(x_1, \ldots, x_n) = e$
- Expressions
  - $c_1 \mid \ldots \mid c_n$
  - x
  - if b then e_1 else e_2
  - $f(e_1, \ldots, e_n)$
- Booleans
  - true
  - false
  - $e_1 = e_2$
  - $\neg b$
  - $b_1 \wedge b_2$
  - $b_1 \vee b_2$

Computing Predicates: Booleans

- $[\ c_i = c_i\ ] \Rightarrow$ true
- $[\ c_i = c_j\ ] \Rightarrow$ false if $i \neq j$
- $[\ x_i = c_j\ ] \Rightarrow \alpha_{i,j}$

- $[\ \neg\ b\ ] \Rightarrow \neg\ [\ b\ ]$
- $[\ b_1 \wedge b_2\ ] \Rightarrow [\ b_1\ ] \wedge [\ b_2\ ]$
- $[\ b_1 \vee b_2\ ] \Rightarrow [\ b_1\ ] \vee [\ b_2\ ]$

Computing Predicates: Booleans

- $[\ e = e'\ ] \Rightarrow \bigvee_i [\ e = c_i\ ] \wedge [\ e' = c_i\ ]$
- And some more rules …

- If f is undefined
  $[\ f^p(e_1, \ldots, e_n) = c_i\ ] \Rightarrow \beta_{fp,i}$

- If $f(x_1, \ldots, x_n) = e$ then
  $[\ f^p(e_1, \ldots, e_n) = c_i\ ] \Rightarrow \gamma_{fp,i}$

A System of Equations

$$\gamma_1 = E_1(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$
$$\gamma_2 = E_k(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$
$$\ldots$$
$$\gamma_k = E_k(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$

Now What?

- We want to solve these equations

- But how?
  - The γ's are bound
  - The α's correspond to function inputs
    - We want the answer in terms of α's
  - But the β's are not observable outside of their corresponding functions

An Insight

- In practice we care only about answering *may* and *must* queries
  - We don't care about arbitrary solutions of the equations

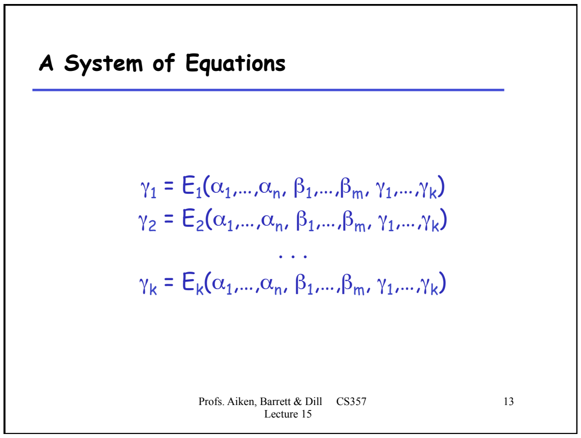
- We want either
  - the *strongest necessary condition* or
  - the *weakest sufficient condition*

  for this system of equations without the β's

Example Revisited

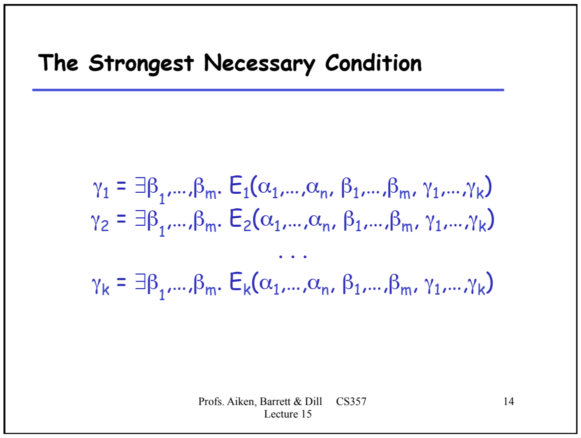
```
void foo(int a) {
   int *t = malloc(…);
   char u = getUserInput();
   if ((t != NULL) || (u == '\n' )) && (a == FLAG)
        return 0;
   else return 1;
}
```

a == FLAG is the strongest necessary condition for returning 0 expressible in terms of foo's function interface.



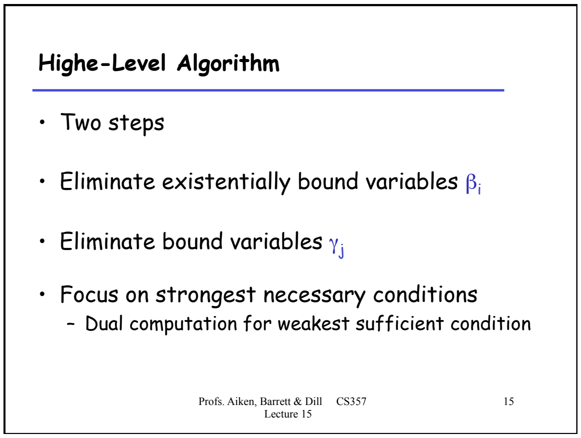
A System of Equations

$$\gamma_1 = E_1(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$
$$\gamma_2 = E_2(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$
$$\dots$$
$$\gamma_k = E_k(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$



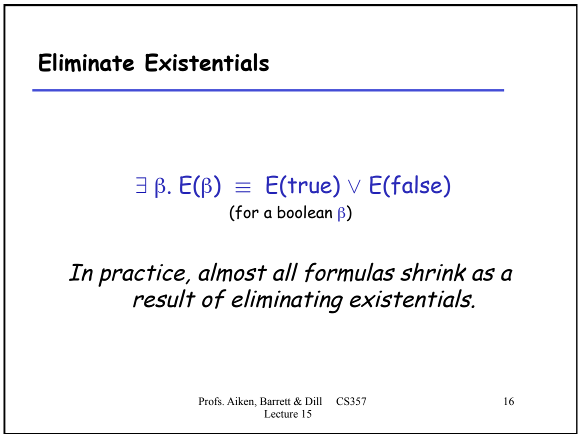
The Strongest Necessary Condition

$$\gamma_1 = \exists\beta_1,\dots,\beta_m.\ E_1(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$
$$\gamma_2 = \exists\beta_1,\dots,\beta_m.\ E_2(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$
$$\dots$$
$$\gamma_k = \exists\beta_1,\dots,\beta_m.\ E_k(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$



Highe-Level Algorithm

- Two steps
- Eliminate existentially bound variables β_i
- Eliminate bound variables γ_j
- Focus on strongest necessary conditions
  - Dual computation for weakest sufficient condition

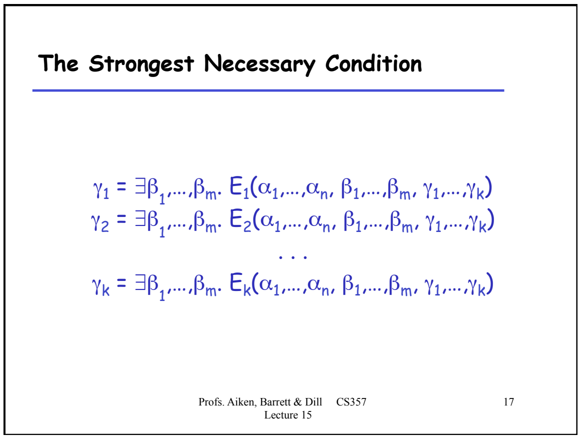


Eliminate Existentials

$$\exists\ \beta.\ E(\beta)\ \equiv\ E(\text{true}) \lor E(\text{false})$$

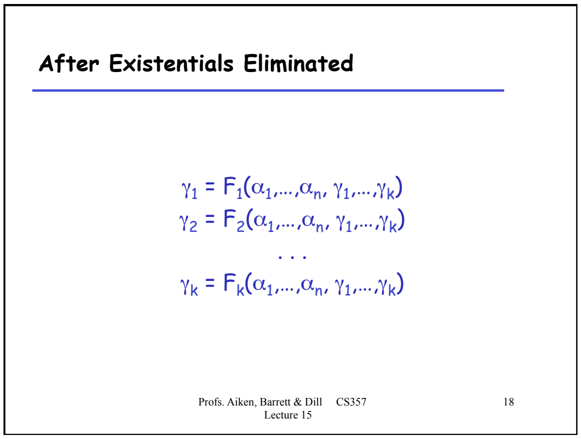
(for a boolean β)

In practice, almost all formulas shrink as a result of eliminating existentials.



The Strongest Necessary Condition

$$\gamma_1 = \exists\beta_1,\dots,\beta_m.\ E_1(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$
$$\gamma_2 = \exists\beta_1,\dots,\beta_m.\ E_2(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$
$$\dots$$
$$\gamma_k = \exists\beta_1,\dots,\beta_m.\ E_k(\alpha_1,\dots,\alpha_n, \beta_1,\dots,\beta_m, \gamma_1,\dots,\gamma_k)$$



After Existentials Eliminated

$$\gamma_1 = F_1(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$
$$\gamma_2 = F_2(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$
$$\dots$$
$$\gamma_k = F_k(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$

Eliminating the Bound Variables

Idea: Compute a fixed point of the equations . . .

$$\gamma_1 = F_1(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$
$$\gamma_2 = F_2(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$
$$. \; . \; .$$
$$\gamma_k = F_k(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$

Problem: The γ's may be negated on the right-hand side . . .

The Trick

- Drive negations in everywhere
  - Negation normal form
  - Negations appear only in literals
- Recall

$$\gamma_{fp,i} \equiv f^p(\dots) = c_i$$

- Use equivalence

$$\neg\gamma_{fp,i} = \bigvee_{i \neq j} \gamma_{fp,j}$$

Solving the Equations

$$\gamma_1 = G_1(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$
$$\gamma_2 = G_2(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$
$$. \; . \; .$$
$$\gamma_k = G_k(\alpha_1,\dots,\alpha_n, \gamma_1,\dots,\gamma_k)$$

Now monotone in the γ's . . .

Iterative Solution

$$\gamma_{i,0} = \text{true} \quad 1 \leq i \leq n$$

$$\gamma_{1,j} = G_1(\alpha_1,\dots,\alpha_n, \gamma_{1,j-1},\dots,\gamma_{k,j-1})$$
$$\gamma_{2,j} = G_2(\alpha_1,\dots,\alpha_n, \gamma_{1,j-1},\dots,\gamma_{k,j-1})$$
$$. \; . \; .$$
$$\gamma_{k,j} = G_k(\alpha_1,\dots,\alpha_n, \gamma_{1,j-1},\dots,\gamma_{k,j-1})$$

A finite ascending chain, and therefore, a fixed point.

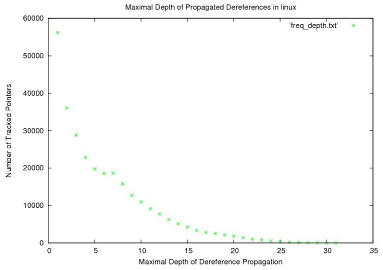
An Example Problem

What is the interprocedural path-sensitive condition under which a pointer will be dereferenced?

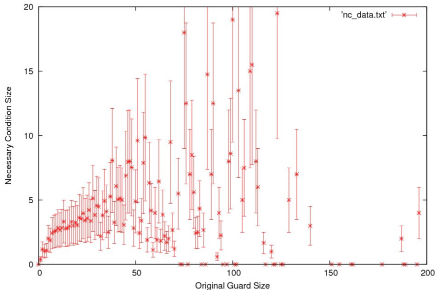
Experiment

- Computed the SNC and WSC for every pointer dereference in Linux
- Two versions of experiment:
  - Condition at each actual dereference, that the dereference actually takes place
    - i.e., that line of code is reached
  - Condition for each pointer source that it will be dereferenced

Procedure Depth of Dereferenced Pointers



Original vs. Necessary Condition



But We're Cheating

- Functions
  $f(x_1, \ldots, x_n) = e$
- Expressions
  $c_1 \mid \ldots \mid c_n$
  x
  if b then e_1 else e_2
  $f(e_1, \ldots, e_n)$
- Booleans
  true
  false
  $e_1 = e_2$
  $\neg b$
  $b_1 \wedge b_2$
  $b_1 \vee b_2$

What To Do?

- Consider

  if (e) then A else B

- Want to analyze this as

  $[e] \Rightarrow [A] \wedge \neg[e] \Rightarrow [B]$

- But $\neg[e]$ is not an overapproximation

Picture: Over- and Underapproximations

The Solution

- Compute an over- and underapproximation for everything!
  - A necessary and sufficient condition
  - *Bracketing constraints*

$$F \equiv (O, U)$$

Algebra of Bracketing Constraints

$$(O_1,U_1) \wedge (O_2,U_2) \equiv (O_1 \wedge O_2, U_1 \wedge U_2)$$

$$(O_1,U_1) \vee (O_2,U_2) \equiv (O_1 \vee O_2, U_1 \vee U_2)$$

$$\neg(O,U) \equiv (\neg U, \neg O)$$

Flashback

- Consider

  if (e) then A else B

- Want to analyze this as

  $[e] \Rightarrow [A] \wedge \neg[e] \Rightarrow [B]$

- And now this is just fine if the formulas are bracketing constraints.

Recall the Systems of Equations

- Necessary condition

$$\gamma_1 = \exists\beta_1,\ldots,\beta_m.\ E_1(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$
$$\gamma_2 = \exists\beta_1,\ldots,\beta_m.\ E_2(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$
$$\ldots$$
$$\gamma_k = \exists\beta_1,\ldots,\beta_m.\ E_k(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$

- Sufficient condition

$$\gamma_1 = \forall\beta_1,\ldots,\beta_m.\ E_1(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$
$$\gamma_2 = \forall\beta_1,\ldots,\beta_m.\ E_2(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$
$$\ldots$$
$$\gamma_k = \forall\beta_1,\ldots,\beta_m.\ E_k(\alpha_1,\ldots,\alpha_n, \beta_1,\ldots,\beta_m, \gamma_1,\ldots,\gamma_k)$$

- These were two separate systems
  - But now they are mutually recursive because $\neg(O,U) \equiv (\neg U, \neg O)$

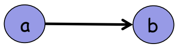
So What?

- Using bracketing constraints, computing underapproximations is not optional
  - Because they are used to define the negation of an overapproximation
- Just results in a bigger system of mutually recursive equations

---

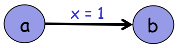
Points-to Graph

*a = b



Points-to Graph in Saturn

if (x = 1) *a = b



What About Aggregate Data Structures?

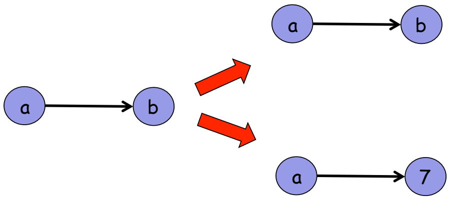
- Use *summary nodes*
- Nodes that represent more than one heap location

Points-To Graphs with Summary Nodes

- *Every* location in a may point to *any* location in b

Updates to Arrays: Case Split the World

a[3] = 7



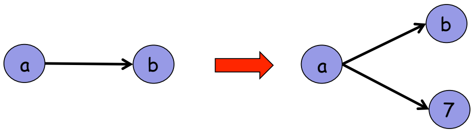
Elements of a except a[3]

a[3]

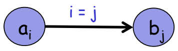
When a node has only one element we can do strong updates

Updates to Arrays: Weak Updates

a[3] = 7



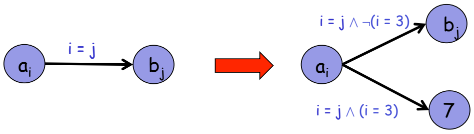
New Representation: Add Index Variables



i and j are variables ranging over the indices of the array.

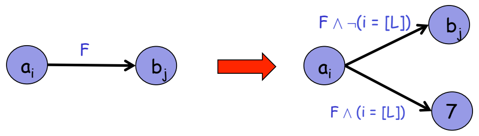
Updates to Arrays: Fluid Updates

a[3] = 7



Fluid Updates in General

a[L] = 7



Three Applications of Bracketing Constraints

- Removing unknowns
- Sound path-sensitive analysis of conditionals
  - When we can't analyze the tests exactly
- Fluid updates
  - Avoid splitting the world eagerly
  - Let the constraint solver do it