Over- and Underapproximations in Program Analysis

A Possible Problem

```plaintext
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

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

Condition under which 0 is returned:

```
((t != NULL) || (a == \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
  ```plaintext
  f(x_1, \ldots, x_n) = e
  ```
- Expressions
  ```plaintext
  c_1 \mid \ldots \mid c_n
  ```
  ```plaintext
  x
  ```
  ```plaintext
  if b then e_1 else e_2
  ```
  ```plaintext
  f(e_1, \ldots, e_n)
  ```
- Booleans
  ```plaintext
  true
  ```
  ```plaintext
  false
  ```
  ```plaintext
  e_1 = e_2
  ```
  ```plaintext
  \neg b
  ```
  ```plaintext
  b_1 \land b_2
  ```
  ```plaintext
  b_1 \lor b_2
  ```
Computing Predicates: Booleans

• \([ c_i = c_j ] \Rightarrow \text{true} \)
• \([ c_i = c_j ] \Rightarrow \text{false} \) if \( i \neq j \)
• \([ x_i = c_j ] \Rightarrow \alpha_{ij} \)
• \([- b ] \Rightarrow \neg [ b ] \)
• \([ b_i \land b_j ] \Rightarrow [ b_i ] \land [ b_j ] \)
• \([ b_i \lor b_j ] \Rightarrow [ b_i ] \lor [ b_j ] \)

A System of Equations

\[
\begin{align*}
\gamma_1 &= E_1(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k) \\
\gamma_2 &= E_2(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k) \\
& \quad \vdots \\
\gamma_k &= E_k(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k)
\end{align*}
\]

Now What?

• We want to solve these equations
• But how?
  - The \( \gamma \)'s are bound
  - The \( \alpha \)'s correspond to function inputs
  - We want the answer in terms of \( \alpha \)'s
  - But the \( \beta \)'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 \( \beta \)'s

Example Revisited

```c
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

\[ \gamma_1 = E_1(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k) \]
\[ \gamma_2 = E_2(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k) \]
\[ \vdots \]
\[ \gamma_k = E_k(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k) \]

The Strongest Necessary Condition

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

Highe-Level Algorithm

- Two steps
- Eliminate existentially bound variables \( \beta_i \)
- Eliminate bound variables \( \gamma_j \)
- 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 \( \beta \))

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

The Strongest Necessary Condition

\[ \gamma_1 = F_1(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \gamma_2 = F_2(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \vdots \]
\[ \gamma_k = F_k(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]

After Existentials Eliminated

\[ \gamma_1 = F_1(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \gamma_2 = F_2(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \vdots \]
\[ \gamma_k = F_k(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
Eliminating the Bound Variables

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

\[ \gamma_1 = F_1(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \gamma_2 = F_2(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \vdots \]
\[ \gamma_k = F_k(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]

Problem: The \( \gamma \)'s may be negated on the right-hand side...

The Trick

- Drive negations in everywhere
  - Negation normal form
  - Negations appear only in literals
- Recall
  \[ \gamma_{fp,i} = f_i(\ldots) \leq c_i \]
- Use equivalence
  \[ \neg \gamma_{fp,i} = \bigvee_{i \neq j} \gamma_{fp,j} \]

Solving the Equations

\[ \gamma_1 = G_1(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \gamma_2 = G_2(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]
\[ \vdots \]
\[ \gamma_k = G_k(\alpha_1, \ldots, \alpha_n, \gamma_1, \ldots, \gamma_k) \]

Now monotone in the \( \gamma \)'s...

Iterative Solution

\[ \gamma_{0i} = \text{true} \quad 1 \leq i \leq n \]
\[ \gamma_{1j} = G_j(\alpha_1, \ldots, \alpha_n, \gamma_{1-1}, \ldots, \gamma_{k-1}) \]
\[ \gamma_{2j} = G_j(\alpha_1, \ldots, \alpha_n, \gamma_{1-1}, \ldots, \gamma_{k-1}) \]
\[ \vdots \]
\[ \gamma_{kj} = G_j(\alpha_1, \ldots, \alpha_n, \gamma_{k-1}, \ldots, \gamma_{j-1}) \]

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
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
  \[ true \]
  \[ false \]
  \[ e_1 = e_2 \]
  \[ ¬b \]
  \[ b_1 \land b_2 \]
  \[ b_1 \lor b_2 \]

What To Do?

- Consider
  \[ if\ (e)\ then\ A\ else\ B \]

- Want to analyze this as
  \[ [e] \Rightarrow [A] \land [\neg e] \Rightarrow [B] \]

- But \([\neg e]\) is not an overapproximation

The Solution

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

\[ F = (O, U) \]
Algebra of Bracketing Constraints

\[(O_1, U_1) \land (O_2, U_2) = (O_1 \land O_2, U_1 \land U_2)\]
\[(O_1, U_1) \lor (O_2, U_2) = (O_1 \lor O_2, U_1 \lor U_2)\]
\[\neg(O, U) = (-U, -O)\]

Flashback

- Consider if (e) then A else B
- Want to analyze this as \[e \Rightarrow [A] \land \neg[e] \Rightarrow [B]\]
- And now this is just fine if the formulas are bracketing constraints.

Recall the Systems of Equations

- Necessary condition
  \[\gamma_1 = \forall \beta_1 \ldots \beta_m. E(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k)\]
  \[\gamma_2 = \forall \beta_1 \ldots \beta_m. E(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k)\]
  \[\gamma_k = \forall \beta_1 \ldots \beta_m. E(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k)\]
- Sufficient condition
  \[\gamma_1 \leq \beta_1 \ldots \beta_m. E(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k)\]
  \[\gamma_2 \leq \beta_1 \ldots \beta_m. E(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k)\]
  \[\gamma_k \leq \beta_1 \ldots \beta_m. E(\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m, \gamma_1, \ldots, \gamma_k)\]
- These were two separate systems
  - But now they are mutually recursive because \[\neg(O, U) = (-U, -O)\]

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

\[\ast a = b\]

Points-to Graph in Saturn

\[\text{if (x = 1) } \ast a = b\]
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 → b
```

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

```
a[3] → 7
```

Updates to Arrays: Weak Updates

```
a[3] = 7
```

```
a → b
```

```
a → 7
```

Updates to Arrays: Fluid Updates

```
a[3] = 7
```

```
a → b
```

```
i ≠ j ∧ \neg (i = 3)
```

```
i = j ∧ (i = 3)
```

New Representation: Add Index Variables

```
i, j
```

```
i and j are variables ranging over the indices of the array.
```
**Fluid Updates in General**

\[ a[L] = 7 \]

\[ F \land (i = [L]) \]

**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