Constraint-Based Analysis in the Presence of Uncertainty and Imprecision

Isil Dillig and Thomas Dillig
Joint work with Alex Aiken
Computer Science Department
Stanford University

February 19, 2009

Motivation

- When we reason about programs statically, uncertainty and imprecision come up everywhere.

Motivation

- When we reason about programs statically, uncertainty and imprecision come up everywhere.
  - Uncertainty: We often do not (or cannot) model every aspect of the environment the program executes in

Motivation

- When we reason about programs statically, uncertainty and imprecision come up everywhere.
    - Uncertainty: We often do not (or cannot) model every aspect of the environment the program executes in
    - Imprecision: Any analysis is necessarily based on some abstraction of the program

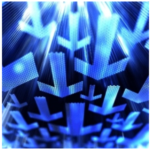
Uncertainty

- User Input

```
if(getUserInput() == 'y') return true;
else return false;
```

Uncertainty

- User Input
- Network data



```
char buf[1024];
recv(socket,buf,1024,0);
struct data* d = (struct data*) buf;
```

Uncertainty

- User Input
- Network data
- System state

```
int* p = malloc(sizeof(int)*num_elems);
if(p == NULL) exit(1);
```

Uncertainty

- User Input
- Network data
- System state
- Many more
  - e.g., calling an unknown function, thread scheduling, . . .

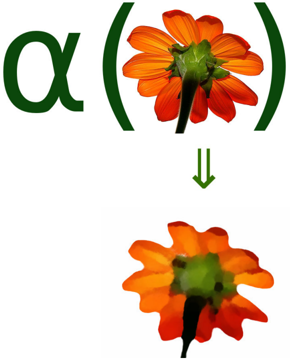
Uncertainty

- User Input
- Network data
- System state
- Many more...

All of these appear as non-deterministic environment choices

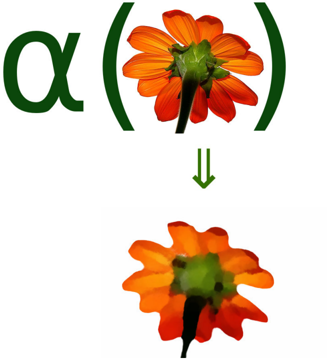
Imprecision

- In constrast to uncertainty, imprecision arises from the abstraction intentionally chosen by the analysis designer



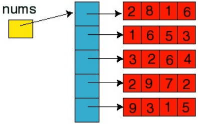
Imprecision

- In constrast to uncertainty, imprecision arises from the abstraction intentionally chosen by the analysis designer
- But imprecision results in similar consequences as uncertainty



Imprecision

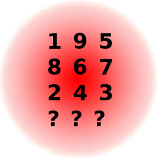
- Many program analyis systems do not reason about unbounded data structures or abstract data types




```
int elem = array[i];
assert(elem != -1);
```

Imprecision

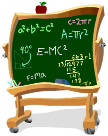
- Many program analyis systems do not reason about unbounded data structures or abstract data types



```
int elem =  ;
assert(elem != -1);
```

Imprecision

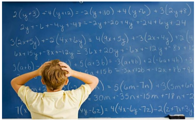
- Many program analyis systems do not reason about unbounded data structures or abstract data types
- Many systems do not track "complicated" arithmetic




```
if(COEF*a*b+MIN_SIZE >= MAX)
     return -1;
```

Imprecision

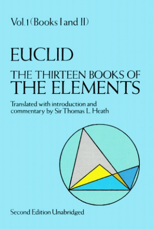
- Many program analyis systems do not reason about unbounded data structures or abstract data types
- Many systems do not track “complicated” arithmetic



`if(`  `>= MAX)`
`    return -1;`

Imprecision

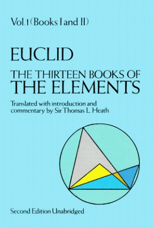
- Many program analyis systems do not reason about unbounded data structures or abstract data types
- Many systems do not track “complicated” arithmetic
- Many systems cannot infer complicated loop invariants




```
int compute_gcd(int a, int b) {
    while(b!=0) {
     int t = a;
     a = b;
     b = t % b;
   }
   return a;
}
assert(x%compute_gcd(x,y) == 0);
```

Imprecision

- Many program analyis systems do not reason about unbounded data structures or abstract data types
- Many systems do not track “complicated” arithmetic
- Many systems cannot infer complicated loop invariants

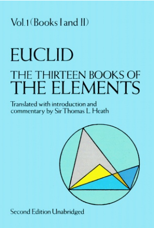



```
int compute_gcd(int a, int b) {
```

```
}
assert(x%compute_gcd(x,y) == 0);
```

Imprecision

- Many program analyis systems do not reason about unbounded data structures or abstract data types
- Many systems do not track "complicated" arithmetic
- Many systems cannot infer complicated loop invariants




```
int compute_gcd(int a, int b)
{
```

```
}
```

`assert(x%`  `== 0);`

Imprecision

- Many program analyis systems do not reason about unbounded data structures or abstract data types
- Many systems do not track "complicated" arithmetic
- Many systems cannot infer complicated loop invariants

Sources of imprecision appear as non-deterministic environment choices

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
- For example, whenever there is a call to `getUserInput()`, introduce a fresh variable β.

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
- For example, whenever there is a call to `getUserInput()`, introduce a fresh variable β.

SAT($\beta =$ 'y') ?

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
- For example, whenever there is a call to `getUserInput()`, introduce a fresh variable β.

SAT($\beta =$ 'y') ?   Of course!

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
- For example, whenever there is a call to `getUserInput()`, introduce a fresh variable β.

SAT($\beta \neq$ 'y') ?

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
- For example, whenever there is a call to `getUserInput()`, introduce a fresh variable β.

SAT($\beta \neq$ 'y') ?   Of course!

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
- For example, whenever there is a call to `getUserInput()`, introduce a fresh variable β.

VALID($\beta =$ 'y') ?

Modeling Choice in Constraint Based Analyses

- In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
- For example, whenever there is a call to `getUserInput()`, introduce a fresh variable β.

VALID($\beta =$ 'y') ? Of course not!

Modeling Choice in Constraint Based Analyses

- Unfortunately, the use of choice variables may introduce two problems:

Modeling Choice in Constraint Based Analyses

- Unfortunately, the use of choice variables may introduce two problems:
  - Theoretical: It is not clear how to solve recursive constraints containing choice variables.

Modeling Choice in Constraint Based Analyses

- Unfortunately, the use of choice variables may introduce two problems:
  - Theoretical: It is not clear how to solve recursive constraints containing choice variables.
  - Practical: The number of choice variables is proportional to the size of the analyzed program.

Large formulas $\Rightarrow$ Poor scalability

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

When does queryUser return true?

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

Given an arbitrary argument α, what is the constraint $\Pi_{\alpha,\text{true}}$ under which queryUser returns true?

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

$$\boxed{\Pi_{\alpha,\mathsf{true}} = ?}$$

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

$$\Pi_{\alpha,\mathsf{true}} = (\alpha = \mathsf{true}) \wedge ?$$

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

$$\Pi_{\alpha,\mathsf{true}} = ((\alpha = \mathsf{true}) \wedge (\beta = \text{'y'} \vee ?))$$

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled) ;
}
```

$$\Pi_{\alpha,\mathsf{true}} = ((\alpha = \mathsf{true}) \wedge (\beta = \text{'y'} \vee (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta'/\beta])))$$

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

$$\Pi_{\alpha,\mathsf{true}} \;=\; ((\alpha = \mathsf{true}) \;\wedge\; (\beta = \text{'y'} \;\vee\; (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\mathsf{true}}\, [\mathsf{true}/\alpha]\, [\beta'/\beta])))$$

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

$$\Pi_{\alpha,\mathsf{true}} = ((\alpha = \mathsf{true}) \wedge (\beta = \text{'y'} \vee (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha]\,[\beta'/\beta])))$$

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

$$\Pi_{\alpha,\mathsf{true}} \;=\; ((\alpha = \mathsf{true}) \;\wedge\; (\beta = \text{'y'} \;\vee\; (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta'/\beta])))$$

Recursive Constraint Example

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

$$\Pi_{\alpha,\text{true}} \;=\; ((\alpha = \text{true}) \;\wedge\; (\beta = \text{'y'} \;\vee\; (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\text{true}}[\text{true}/\alpha][\beta'/\beta])))$$

Recursive Constraint Example

- If we solve this constraint naively using standard fix-point computation, we get:

$$\begin{aligned}\Pi_{\alpha,\mathtt{true}} = \quad & (\alpha = \mathtt{true}) \wedge (\beta = \mathtt{'y'} \vee (\neg(\beta = \mathtt{'n'}) \wedge \\ & \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta'/\beta]))\end{aligned}$$

Recursive Constraint Example

- If we solve this constraint naively using standard fix-point computation, we get:

$$\begin{aligned}\Pi_{\alpha,\texttt{true}} = \quad & (\alpha = \texttt{true}) \wedge (\beta = '\texttt{y}' \vee (\neg(\beta = '\texttt{n}')))\wedge \\ & (\texttt{true} = \texttt{true}) \wedge (\beta' = '\texttt{y}' \vee \neg(\beta' = '\texttt{n}')\wedge \\ & \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta''/\beta']\end{aligned}$$

Recursive Constraint Example

- If we solve this constraint naively using standard fix-point computation, we get:

$$\begin{aligned}\Pi_{\alpha,\texttt{true}} = \quad & (\alpha = \texttt{true}) \wedge (\beta = '\texttt{y}' \vee (\neg(\beta = '\texttt{n}')))\wedge \\ & (\texttt{true} = \texttt{true}) \wedge (\beta' = '\texttt{y}' \vee \neg(\beta' = '\texttt{n}')\wedge \\ & {\color{red}(\texttt{true} = \texttt{true}) \wedge (\beta'' = '\texttt{y}' \vee \neg(\beta'' = '\texttt{n}')}\wedge \\ & \ldots\end{aligned}$$

Recursive Constraint Example

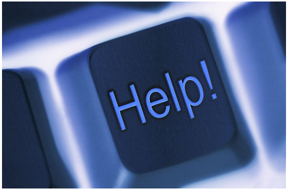
- If we solve this constraint naively using standard fix-point computation, we get:

$$\begin{aligned}\Pi_{\alpha,\mathtt{true}} = \quad & (\alpha = \mathtt{true}) \wedge (\beta = \mathtt{'y'} \vee (\neg(\beta = \mathtt{'n'})))\wedge \\ & (\mathtt{true} = \mathtt{true}) \wedge (\beta' = \mathtt{'y'} \vee \neg(\beta' = \mathtt{'n'})\wedge \\ & (\mathtt{true} = \mathtt{true}) \wedge (\beta'' = \mathtt{'y'} \vee \neg(\beta'' = \mathtt{'n'})\wedge \\ & \ldots\end{aligned}$$

- It is not clear how to solve recursive constraints involving choice variables.

Scalability

Even if we had a way of solving such recursive constraints, choice variables remain a source of scalability problems, even for reasonably sized programs.



Scalability Problem Example

```
Key * key_new_private(int type) {
  Key *k = key_new(type);
  switch (type) {
    case KEY_RSA1:
    case KEY_RSA:
      if ((k->rsa->d = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->iqmp = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->q = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->p = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmq1 = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmp1 = BN_new()) == NULL) fatal("BN_new failed");
      break;
    case KEY_DSA:
      if ((k->dsa->priv_key = BN_new()) == NULL) fatal("BN_new failed");
    default:
      break;  }
  return k;  }
```

Scalability Problem Example

```
Key * key_new_private(int type) {
  Key *k = key_new(type);
  switch (type) {
    case KEY_RSA1:
    case KEY_RSA:
      if ((k->rsa->d = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->iqmp = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->q = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->p = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmq1 = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmp1 = BN_new()) == NULL) fatal("BN_new failed");
      break;
    case KEY_DSA:
      if ((k->dsa->priv_key = BN_new()) == NULL) fatal("BN_new failed");
    default:
      break;  }
  return k;  }
```

Assume KEY_RSA1, KEY_RSA, and KEY_DSA are #define'd as 1, 2 and 3 respectively.

Scalability Problem Example

```
Key * key_new_private(int type) {
  Key *k = key_new(type);
  switch (type) {
    case  1:
    case  2:
      if ((k->rsa->d = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->iqmp = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->q = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->p = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmq1 = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmp1 = BN_new()) == NULL) fatal("BN_new failed");
      break;
    case  3:
      if ((k->dsa->priv_key = BN_new()) == NULL) fatal("BN_new failed");
    default:
      break;  }
  return k;  }
```

Scalability Problem Example

```
Key * key_new_private(int type) {
  Key *k = key_new(type);
  switch (type) {
    case  1:
    case  2:
      if ((k->rsa->d = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->iqmp = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->q = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->p = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmq1 = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmp1 = BN_new()) == NULL) fatal("BN_new failed");
      break;
    case  3:
      if ((k->dsa->priv_key = BN_new()) == NULL) fatal("BN_new failed");
    default:
      break;  }
  return k;  }
```

What is the constraint under which `key_new_private` successfully returns a new key?

Scalability Problem Example

- Denoting the argument of `key_new_private` by α, let us slice the relevant part of the function:

Scalability Problem Example

- Denoting the argument of key_new_private by α, let us slice the relevant part of the function:

```
key_new_private(α) {
  if (α == 1 || α == 2 ) {
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
  }
  if (α == 3)
      if (BN_new()) == NULL) /* fail */
  /* success */
}
```

Scalability Problem Example

- Denoting the argument of `key_new_private` by α, let us slice the relevant part of the function:

```
key_new_private(α) {
  if (α == 1 || α == 2 ) {
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
  }
  if (α == 3)
      if (BN_new()) == NULL) /* fail */
  /* success */
}
```

- Here, `BN_NEW` is a `malloc` wrapper; hence, its return value should be treated as non-deterministic environment choice

Scalability Problem Example

- Denoting the argument of `key_new_private` by α, let us slice the relevant part of the function:

```
key_new_private(α) {
  if (α == 1 || α == 2 ) {
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
      if (BN_new() == NULL) /* fail */
  }
  if (α == 3)
      if (BN_new()) == NULL) /* fail */
  /* success */
}
```

- We replace each call to `BN_NEW` with a fresh choice variable β_i.

Scalability Problem Example

```
key_new_private(α) {
  if (α == 1 || α == 2 ) {
    if (β1 == 0 || β2 == 0 || β3 == 0 || β4 == 0 || β5 == 0)
      /* fail */
  }
  if (α == 3)
      if (β6 == 0) /* fail */
  /* success */
}
```

Scalability Problem Example

```
key_new_private(α) {
  if (α == 1 || α == 2 ) {
    if (β1 == 0 || β2 == 0 || β3 == 0 || β4 == 0 || β5 == 0)
      /* fail */
  }
  if (α == 3)
      if (β6 == 0) /* fail */
  /* success */
}
```

- The condition under which the function succeeds is:

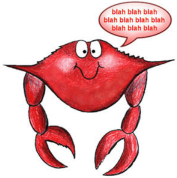
$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0)\\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

Scalability Problem Example

```
key_new_private(α) {
  if (α == 1 || α == 2 ) {
    if (β1 == 0 || β2 == 0 || β3 == 0 || β4 == 0 || β5 == 0)
      /* fail */
  }
  if (α == 3)
      if (β6 == 0) /* fail */
  /* success */
}
```

- The condition under which the function succeeds is:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$



- Very verbose way of stating the success condition!

Scalability Problem Example

- Now consider some calling context of this function:

```
Key* rsa1_key = key_new_private(KEY_RSA1);
Key* rsa_key = key_new_private(KEY_RSA);
Key* dsa_key = key_new_private(KEY_DSA);
/* SUCCESS */
```

Scalability Problem Example

- Now consider some calling context of this function:

```
Key* rsa1_key = key_new_private(KEY_RSA1);
Key* rsa_key = key_new_private(KEY_RSA);
Key* dsa_key = key_new_private(KEY_DSA);
/* SUCCESS */
```

- What is the constraint under which we reach `/*SUCCESS*/`?

Scalability Problem Example

- Now consider some calling context of this function:

```
Key* rsa1_key = key_new_private(KEY_RSA1);
Key* rsa_key = key_new_private(KEY_RSA);
Key* dsa_key = key_new_private(KEY_DSA);
/* SUCCESS */
```

- What is the constraint under which we reach `/*SUCCESS*/`?

$$
\begin{gathered}
(1 \leq 1 \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\
\vee (1 = 3 \wedge \beta_6 \neq 0) \vee 1 \leq 0 \vee 1 \geq 4) \quad \wedge \\
(1 \leq 2 \leq 2 \wedge (\beta'_1 \neq 0 \wedge \beta'_2 \neq 0 \wedge \beta'_3 \neq 0 \wedge \beta'_4 \neq 0 \wedge \beta'_5 \neq 0) \\
\vee (2 = 3 \wedge \beta'_6 \neq 0) \vee 2 \leq 0 \vee 2 \geq 4) \quad \wedge \\
(1 \leq 3 \leq 2 \wedge (\beta''_1 \neq 0 \wedge \beta''_2 \neq 0 \wedge \beta''_3 \neq 0 \wedge \beta''_4 \neq 0 \wedge \beta''_5 \neq 0) \\
\vee (3 = 3 \wedge \beta''_6 \neq 0) \vee 3 \leq 0 \vee 3 \geq 4)
\end{gathered}
$$

Scalability Problem Example

- Now consider some calling context of this function:

```
Key* rsa1_key = key_new_private(KEY_RSA1);
Key* rsa_key = key_new_private(KEY_RSA);
Key* dsa_key = key_new_private(KEY_DSA);
/* SUCCESS */
```

- What is the constraint under which we reach `/*SUCCESS*/`?

$$
\begin{array}{c}
(1 \leq 1 \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\
\vee (1 = 3 \wedge \beta_6 \neq 0) \vee 1 \leq 0 \vee 1 \geq 4) \quad \wedge \\
(1 \leq 2 \leq 2 \wedge (\beta'_1 \neq 0 \wedge \beta'_2 \neq 0 \wedge \beta'_3 \neq 0 \wedge \beta'_4 \neq 0 \wedge \beta'_5 \neq 0) \\
\vee (2 = 3 \wedge \beta'_6 \neq 0) \vee 2 \leq 0 \vee 2 \geq 4) \quad \wedge \\
(1 \leq 3 \leq 2 \wedge (\beta''_1 \neq 0 \wedge \beta''_2 \neq 0 \wedge \beta''_3 \neq 0 \wedge \beta''_4 \neq 0 \wedge \beta''_5 \neq 0) \\
\vee (3 = 3 \wedge \beta''_6 \neq 0) \vee 3 \leq 0 \vee 3 \geq 4)
\end{array}
$$

Conclusion from the Examples

- Introducing choice variables causes problems both with scalability and solving recursive constraints

Conclusion from the Examples

- Introducing choice variables causes problems both with scalability and solving recursive constraints
- It is desirable to eliminate these choice variables from the constraints

Conclusion from the Examples

- Introducing choice variables causes problems both with scalability and solving recursive constraints
- It is desirable to eliminate these choice variables from the constraints
- Idea: Compute an over-approximation of the constraint not containing any choice variables

Strongest Necessary Conditions

- An over-approximation $\lceil \phi \rceil$ of a constraint ϕ not containing choice variables is implied by the original constraint, i.e. $\lceil \phi \rceil$ is a necessary condition.

$$\phi \Rightarrow \lceil \phi \rceil$$

Strongest Necessary Conditions

- An over-approximation $\lceil \phi \rceil$ of a constraint ϕ not containing choice variables is implied by the original constraint, i.e. $\lceil \phi \rceil$ is a necessary condition.

$$\phi \Rightarrow \lceil \phi \rceil$$

- But rather than computing any necessary condition, we want to compute the strongest necessary condition:

$$\forall \phi'.((\phi \Rightarrow \phi') \Rightarrow (\lceil \phi \rceil \Rightarrow \phi'))$$

Strongest Necessary Conditions

- An over-approximation $\lceil\phi\rceil$ of a constraint ϕ not containing choice variables is implied by the original constraint, i.e. $\lceil\phi\rceil$ is a necessary condition.

$$\phi \Rightarrow \lceil\phi\rceil$$

- But rather than computing any necessary condition, we want to compute the strongest necessary condition:

$$\forall\phi'.((\phi \Rightarrow \phi') \Rightarrow (\lceil\phi\rceil \Rightarrow \phi'))$$

- Because strongest necessary condition $\lceil\phi\rceil$ preserves the satisfiability of ϕ:

$$\mathrm{SAT}(\phi) \Leftrightarrow \mathrm{SAT}(\lceil\phi\rceil)$$

SNC Example 1

- Consider the constraint from `key_new_private`:

$$
\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}
$$

SNC Example 1

- Consider the constraint from `key_new_private`:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary condition for this formula is just true.

SNC Example 1

```
Key * key_new_private(int type) {
  Key *k = key_new(type);
  switch (type) {
    case KEY_RSA1:
    case KEY_RSA:
      if ((k->rsa->d = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->iqmp = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->q = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->p = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmq1 = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmp1 = BN_new()) == NULL) fatal("BN_new failed");
      break;
    case KEY_DSA:
      if ((k->dsa->priv_key = BN_new()) == NULL) fatal("BN_new failed");
    default:
      break;  }
  return k;  }
```

- key_new_private MAY successfully return a valid key no matter what the type of the requested cryptographic key is.

SNC Example 2

- Consider the constraint from the `queryUser` function:

$$\Pi_{\alpha,\mathsf{true}} \;=\; ((\alpha = \mathsf{true}) \;\wedge\; (\beta = \text{'y'} \;\vee\; (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta'/\beta])))$$

SNC Example 2

- Consider the constraint from the `queryUser` function:

$$\Pi_{\alpha,\mathsf{true}} \;=\; ((\alpha = \mathsf{true}) \;\wedge\; (\beta = \text{'y'} \;\vee\; (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta'/\beta])))$$

- The strongest necessary condition for $\Pi_{\alpha,\mathrm{true}}$ is $\alpha = \mathrm{true}$.

SNC Example 2

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

- If `feature_enabled` is true in the calling context, queryUser MAY return true
- If `feature_enabled` is false, queryUser will not return true.

Weakest Sufficient Conditions

- Assuming we have a way of computing the strongest necessary condition in a given theory, are we done?

Weakest Sufficient Conditions

- Assuming we have a way of computing the strongest necessary condition in a given theory, are we done?
- Unfortunately, if we only compute strongest necessary conditions, we can no longer safely negate our constraints...

Weakest Sufficient Conditions

- Assuming we have a way of computing the strongest necessary condition in a given theory, are we done?
- Unfortunately, if we only compute strongest necessary conditions, we can no longer safely negate our constraints...

$$\lceil \neg \phi \rceil \not\Leftrightarrow \neg \lceil \phi \rceil$$

Weakest Sufficient Conditions

- Assuming we have a way of computing the strongest necessary condition in a given theory, are we done?
- Unfortunately, if we only compute strongest necessary conditions, we can no longer safely negate our constraints...

$$\lceil \neg \phi \rceil \not\Leftrightarrow \neg \lceil \phi \rceil$$

- Therefore, we need a dual notion of strongest necessary conditions, i.e. weakest sufficient conditions.

Weakest Sufficient Conditions

- The weakest sufficient condition $\lfloor \phi \rfloor$ of formula ϕ not containing any choice variables satisfies:

$$\begin{array}{ll} (1) & \lfloor \phi \rfloor \Rightarrow \phi \\ (2) & \forall \phi'.((\phi' \Rightarrow \phi) \Rightarrow (\phi' \Rightarrow \lfloor \phi \rfloor)) \end{array}$$

Weakest Sufficient Conditions

- The weakest sufficient condition $\lfloor \phi \rfloor$ of formula ϕ not containing any choice variables satisfies:

$$\begin{array}{ll} (1) & \lfloor \phi \rfloor \Rightarrow \phi \\ (2) & \forall \phi'.((\phi' \Rightarrow \phi) \Rightarrow (\phi' \Rightarrow \lfloor \phi \rfloor)) \end{array}$$

- Just as strongest necessary conditions preserve satisfiability, weakest sufficient conditions preserve validity:

$$\mathrm{VALID}(\phi) \Leftrightarrow \mathrm{VALID}(\lfloor \phi \rfloor)$$

WSC Example 1

- Consider the constraint from key_new_private:

$$\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}$$

WSC Example 1

- Consider the constraint from `key_new_private`:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The weakest sufficient condition for this formula is $\alpha \leq 0 \vee \alpha \geq 4$.

WSC Example 1

```
Key * key_new_private(int type) {
  Key *k = key_new(type);
  switch (type) {
    case KEY_RSA1:
    case KEY_RSA:
      if ((k->rsa->d = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->iqmp = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->q = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->p = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmq1 = BN_new()) == NULL) fatal("BN_new failed");
      if ((k->rsa->dmp1 = BN_new()) == NULL) fatal("BN_new failed");
      break;
    case KEY_DSA:
      if ((k->dsa->priv_key = BN_new()) == NULL) fatal("BN_new failed");
    default:
      break;  }
  return k;  }
```

- key_new_private MUST successfully return a valid key if the type of the requested cryptographic key is neither KEY_RSA1, nor KEY_RSA, nor KEY_DSA

WSC Example 2

- Consider the constraint from the `queryUser` function:

$$\Pi_{\alpha,\mathsf{true}} = ((\alpha = \mathsf{true}) \land (\beta = \text{'y'} \lor (\beta \neq \text{'n'} \land \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta'/\beta])))$$

WSC Example 2

- Consider the constraint from the `queryUser` function:

$$\Pi_{\alpha,\mathsf{true}} \;=\; ((\alpha = \mathsf{true}) \;\wedge\; (\beta = \text{'y'} \;\vee\; (\beta \neq \text{'n'} \wedge \Pi_{\alpha,\mathsf{true}}[\mathsf{true}/\alpha][\beta'/\beta])))$$

- The weakest sufficient condition for this formula is false.

WSC Example 2

```
bool queryUser(bool featureEnabled) {
   if(!featureEnabled) return false;
   char userInput = getUserInput();
   if(userInput == 'y') return true;
   if(userInput=='n') return false;
   printf("Input must be y or n!  Please try again");
   return queryUser(featureEnabled);
}
```

- No condition on `feature_enabled` is sufficient to guarantee `queryUser` will return true.
- Hence, the weakest sufficient condition is false.

Negation Revisited

- By having pairs of necessary and sufficient conditions, $(\lceil \phi \rceil, \lfloor \phi \rfloor)$, we can now make negation work:

Negation Revisited

- By having pairs of necessary and sufficient conditions, $(\lceil \phi \rceil, \lfloor \phi \rfloor)$, we can now make negation work:

$$\neg(\lceil \phi \rceil, \lfloor \phi \rfloor) = (\neg \lfloor \phi \rfloor, \neg \lceil \phi \rceil)$$

- The strongest necessary condition for $\neg\phi$ is given by the negation of its weakest sufficient condition, $\neg\lfloor \phi \rfloor$.

- Similarly, the weakest sufficient condition for $\neg\phi$ is given by the negation of ϕ's strongest necessary condition, $\neg\lceil \phi \rceil$.

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\mathrm{true}, \alpha \leq 0 \vee \alpha \geq 4)$$

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\text{true}, \alpha \leq 0 \vee \alpha \geq 4)$$

- Strongest necessary and weakest sufficient conditions for failure:

$$(?, ?)$$

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\text{true}, \alpha \leq 0 \vee \alpha \geq 4)$$

- Strongest necessary and weakest sufficient conditions for failure:

$$(\neg(\alpha \leq 0 \vee \alpha \geq 4), ?)$$

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\text{true}, \alpha \leq 0 \vee \alpha \geq 4)$$

- Strongest necessary and weakest sufficient conditions for failure:

$$(1 \leq \alpha \leq 3, ?)$$

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\mathrm{true}, \alpha \leq 0 \vee \alpha \geq 4)$$

- Strongest necessary and weakest sufficient conditions for failure:

$$(1 \leq \alpha \leq 3, \neg \mathrm{true})$$

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \le \alpha \le 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \le 0 \vee \alpha \ge 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\text{true}, \alpha \le 0 \vee \alpha \ge 4)$$

- Strongest necessary and weakest sufficient conditions for failure:

$$(1 \le \alpha \le 3, \text{false})$$

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\text{true}, \alpha \leq 0 \vee \alpha \geq 4)$$

- Strongest necessary and weakest sufficient conditions for failure:

$$(1 \leq \alpha \leq 3, \text{false})$$

- Nothing guarantees `key_new_private` will fail; i.e. weakest sufficient condition is false.

Negation Example

- Consider once more the constraint:

$$\boxed{\begin{array}{c}(1 \leq \alpha \leq 2 \wedge (\beta_1 \neq 0 \wedge \beta_2 \neq 0 \wedge \beta_3 \neq 0 \wedge \beta_4 \neq 0 \wedge \beta_5 \neq 0) \\ \vee(\alpha = 3 \wedge \beta_6 \neq 0) \vee \alpha \leq 0 \vee \alpha \geq 4)\end{array}}$$

- The strongest necessary and weakest sufficient conditions for success:

$$(\text{true}, \alpha \leq 0 \vee \alpha \geq 4)$$

- Strongest necessary and weakest sufficient conditions for failure:

$$(1 \leq \alpha \leq 3, \text{false})$$

- Requested key must have type `KEY_RSA1`, `KEY_RSA`, or `KEY_DSA` for function to fail.

What Have We Done So Far?

- We identified a special class of variables, called choice variables that model uncertainty and imprecision in constraint-based analysis.

What Have We Done So Far?

- We identified a special class of variables, called choice variables that model uncertainty and imprecision in constraint-based analysis.
- We argued that computing pairs of strongest necessary and weakest sufficient conditions not containing choice variables allows us:

What Have We Done So Far?

- We identified a special class of variables, called choice variables that model uncertainty and imprecision in constraint-based analysis.

- We argued that computing pairs of strongest necessary and weakest sufficient conditions not containing choice variables allows us:

  - ✓ to overcome termination problems

What Have We Done So Far?

- We identified a special class of variables, called choice variables that model uncertainty and imprecision in constraint-based analysis.

- We argued that computing pairs of strongest necessary and weakest sufficient conditions not containing choice variables allows us:

  - ✓ to overcome termination problems
  - ✓ to mitigate scalability problems

What Have We Done So Far?

- We identified a special class of variables, called choice variables that model uncertainty and imprecision in constraint-based analysis.

- We argued that computing pairs of strongest necessary and weakest sufficient conditions not containing choice variables allows us:

  - ✓ to overcome termination problems
  - ✓ to mitigate scalability problems
  - ✓ to negate constraints in a sound way

What Have We Done So Far?

- We identified a special class of variables, called choice variables that model uncertainty and imprecision in constraint-based analysis.

- We argued that computing pairs of strongest necessary and weakest sufficient conditions not containing choice variables allows us:

  - ✓ to overcome termination problems
  - ✓ to mitigate scalability problems
  - ✓ to negate constraints in a sound way
  - ✓ and preserve satisfiability and validity

What Have We *Not* Done So Far?

We have not shown how to compute strongest necessary and weakest sufficient conditions in any specific theory

Rest of This Talk

- We show how to compute strongest necessary and weakest sufficient conditions for a system of recursive constraints representing the exact path- and context-sensitive conditions under which a property holds

Rest of This Talk

- We show how to compute strongest necessary and weakest sufficient conditions for a system of recursive constraints representing the exact path- and context-sensitive conditions under which a property holds
- We use these strongest necessary and weakest sufficient conditions to perform sound and complete path- and context-sensitive program analysis for answering may and must queries

Rest of This Talk

- We show how to compute strongest necessary and weakest sufficient conditions for a system of recursive constraints representing the exact path- and context-sensitive conditions under which a property holds

- We use these strongest necessary and weakest sufficient conditions to perform sound and complete path- and context-sensitive program analysis for answering may and must queries

  - Completeness assumes a user-provided finite abstraction

Rest of This Talk

- We show how to compute strongest necessary and weakest sufficient conditions for a system of recursive constraints representing the exact path- and context-sensitive conditions under which a property holds
- We use these strongest necessary and weakest sufficient conditions to perform sound and complete path- and context-sensitive program analysis for answering may and must queries
  - Completeness assumes a user-provided finite abstraction
  - No choice variables

Rest of This Talk

- We show how to compute strongest necessary and weakest sufficient conditions for a system of recursive constraints representing the exact path- and context-sensitive conditions under which a property holds

- We use these strongest necessary and weakest sufficient conditions to perform sound and complete path- and context-sensitive program analysis for answering may and must queries

  - Completeness assumes a user-provided finite abstraction

  - No choice variables $\Rightarrow$ Small formulas

Rest of This Talk

- We show how to compute strongest necessary and weakest sufficient conditions for a system of recursive constraints representing the exact path- and context-sensitive conditions under which a property holds
- We use these strongest necessary and weakest sufficient conditions to perform sound and complete path- and context-sensitive program analysis for answering may and must queries
  - Completeness assumes a user-provided finite abstraction
  - No choice variables $\Rightarrow$ Small formulas $\Rightarrow$ Good scalability

Where Does This Approach Fit In?

There are many proposed techniques for path- and context-sensitive program analysis.

- Model checking tools: Bebop, BLAST, SLAM, ...

Where Does This Approach Fit In?

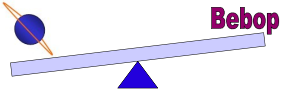
There are many proposed techniques for path- and context-sensitive program analysis.

- Model checking tools: Bebop, BLAST, SLAM, ...
- Lighter-weight static analysis tools: Saturn, ESP, ...

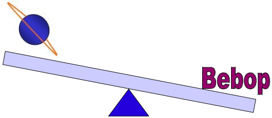
Where Does This Approach Fit In?

There are many proposed techniques for path- and context-sensitive program analysis.

- Model checking tools: Bebop, BLAST, SLAM, ...
- Lighter-weight static analysis tools: Saturn, ESP, ...
- Tradeoff?



The Scalability Scale

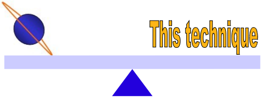


Sound & Complete Scale

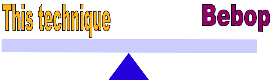
Where Does This Approach Fit In?

There are many proposed techniques for path- and context-sensitive program analysis.

- Model checking tools: Bebop, BLAST, SLAM, ...
- Lighter-weight static analysis tools: Saturn, ESP, ...
- Tradeoff?



The Scalability Scale



Sound & Complete Scale

Contributions

- A sound and complete algorithm for path- and context-sensitive program analysis that scales to multi-million line programs

Contributions

- A sound and complete algorithm for path- and context-sensitive program analysis that scales to multi-million line programs
- Key Insight: 
  - While choice variables are useful within their scoping boundary, they can be eliminated without losing completeness for answering may and must queries about program properties outside of this scoping boundary.

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);
  if(user_input == 'y')
     fclose(f);
}
```

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);
  if(user_input == 'y')
     fclose(f);
}
```

User input is represented by a choice variable

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);
  if(user_input == 'y')
     fclose(f);
}
```

Branch correlation arises from test on choice variable

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
    f = fopen(NEW_FILE_NAME);
  process_file_internal(f);
  if(user_input == 'y')
    fclose(f);
}
```

Correct matching of `fopen()/fclose()`
depends on this branch correlation

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);
  if(user_input == 'y')
     fclose(f);
}
```

Since this user input is not visible in calling contexts of `process_file`, the choice variable is only useful within this scope

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);
  if(user_input == 'y')
     fclose(f);
}
```

If we are interested in answering may and must queries, we can safely eliminate choice variables at their scoping boundaries

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);  /* dereference f */
  if(user_input == 'y')
     fclose(f);
}
```

May the original input file `f` be dereferenced by `process_file`?

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);  /* dereference f */
  if(user_input == 'y')
     fclose(f);
}
```

May the original input file `f` be dereferenced by `process_file`?

YES!

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);  /* dereference f */
  if(user_input == 'y')
     fclose(f);
}
```

Must the original input file `f` be dereferenced by `process_file`?

Choice Variables and Scope

```
void process_file(File* f) {
  printf(``Open new file?\n'');
  char user_input = getUserInput();
  if(user_input == 'y')
     f = fopen(NEW_FILE_NAME);
  process_file_internal(f);  /* dereference f */
  if(user_input == 'y')
     fclose(f);
}
```

Must the original input file `f` be dereferenced by `process_file`?
NO!

Algorithm Outline

1. Set up a recursive constraint system describing the constraints under which each function `f` returns an abstract value C_i

Algorithm Outline

1. Set up a recursive constraint system describing the constraints under which each function `f` returns an abstract value C_i
2. Convert this system to recursive boolean constraints

Algorithm Outline

1. Set up a recursive constraint system describing the constraints under which each function `f` returns an abstract value C_i
2. Convert this system to recursive boolean constraints
3. Eliminate choice variables

Algorithm Outline

1. Set up a recursive constraint system describing the constraints under which each function `f` returns an abstract value C_i
2. Convert this system to recursive boolean constraints
3. Eliminate choice variables
4. Ensure that the system preserves strongest necessary and weakest sufficient conditions under syntactic substitution

Algorithm Outline

1. Set up a recursive constraint system describing the constraints under which each function `f` returns an abstract value C_i
2. Convert this system to recursive boolean constraints
3. Eliminate choice variables
4. Ensure that the system preserves strongest necessary and weakest sufficient conditions under syntactic substitution
5. Solve using standard fixed-point computation

Step 1: Generate constraints

- Set up a recursive system E of constraints describing the constraint Π_{f_i,α,C_j} under which a function f_i, given input α, returns some abstract value C_j:

Step 1: Generate constraints

- Set up a recursive system E of constraints describing the constraint Π_{f_i,α,C_j} under which a function f_i, given input α, returns some abstract value C_j:

$$E = \left[\begin{array}{lcl} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & [\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}][\vec{\beta'}/\vec{\beta}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & [\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}][\vec{\beta'}/\vec{\beta}])] \end{array} \right]$$

Step 1: Generate constraints

- Set up a recursive system E of constraints describing the constraint Π_{f_i,α,C_j} under which a function f_i, given input α, returns some abstract value C_j:

$$E = \begin{bmatrix} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & [\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & [\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \end{bmatrix}$$

- Constraints ϕ_{ij} are boolean combinations of $\alpha = C_i$, $\beta = C_i$, Π_{f_i,α,C_j} and $C_i = C_j$.

Step 1: Generate constraints

- Set up a recursive system E of constraints describing the constraint Π_{f_i,α,C_j} under which a function f_i, given input α, returns some abstract value C_j:

$$E = \begin{bmatrix} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & [\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & [\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \end{bmatrix}$$

- α's represent function inputs, provided by the calling context.

Step 1: Generate constraints

- Set up a recursive system E of constraints describing the constraint Π_{f_i,α,C_j} under which a function f_i, given input α, returns some abstract value C_j:

$$E = \left[\begin{array}{lll} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & [\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & [\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \end{array} \right]$$

- β's represent choice variables. The scope of each β is the function body in which it is introduced.

Step 1: Generate constraints

- Set up a recursive system E of constraints describing the constraint Π_{f_i,α,C_j} under which a function f_i, given input α, returns some abstract value C_j:

$$E = \left[\begin{array}{lcl} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & [\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & [\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \end{array}\right]$$

- Π's on the right hand side result from function calls.

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

- Consider abstract values C_1, C_2, and C_3 such that:

$$C_1 : \{1\}, \quad C_2 : \{2\}, \quad C_3 : Z \backslash \{1, 2\}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

- Consider abstract values C_1, C_2, and C_3 such that:

$$C_1 : \{1\}, \quad C_2 : \{2\}, \quad C_3 : Z \backslash \{1, 2\}$$

- Then,

$$\begin{array}{rcl}\Pi_{f,\alpha,C_1} & = & (\alpha = 1 \lor \beta = 2 \lor \\ & & ((\neg\alpha = 1 \land \neg\beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))\end{array}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

- Consider abstract values C_1, C_2, and C_3 such that:

$$C_1 : \{1\}, \quad C_2 : \{2\}, \quad C_3 : Z \backslash \{1, 2\}$$

- Then,

$$\begin{aligned} \Pi_{f,\alpha,C_1} \quad = \quad & (\alpha = 1 \vee \beta = 2 \vee \\ & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{aligned}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

- Consider abstract values C_1, C_2, and C_3 such that:

$$C_1 : \{1\}, \quad C_2 : \{2\}, \quad C_3 : Z \backslash \{1, 2\}$$

- Then,

$$\begin{aligned} \Pi_{f,\alpha,C_1} \quad = \quad & (\alpha = 1 \lor \beta = 2 \lor \\ & ((\neg\alpha = 1 \land \neg\beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{aligned}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

- Consider abstract values C_1, C_2, and C_3 such that:

$$C_1 : \{1\}, \quad C_2 : \{2\}, \quad C_3 : Z \backslash \{1, 2\}$$

- Then,

$$\begin{aligned} \Pi_{f,\alpha,C_1} \quad = \quad & (\alpha = 1 \vee \beta = 2 \vee \\ & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{aligned}$$

Step 2: Convert to Boolean Constraints

- Convert the previous constraint system to boolean constraints as follows:

$$\begin{array}{lcll} C_i = C_i & \Leftrightarrow & \text{true} & \\ C_i = C_j & \Leftrightarrow & \text{false} & i \neq j \\ v_i = C_j & \Leftrightarrow & v_{ij} & (v_{ij} \text{ fresh}) \end{array}$$

Step 2: Convert to Boolean Constraints

- Convert the previous constraint system to boolean constraints as follows:

$$\begin{array}{lcll} C_i = C_i & \Leftrightarrow & \text{true} & \\ C_i = C_j & \Leftrightarrow & \text{false} & i \neq j \\ v_i = C_j & \Leftrightarrow & v_{ij} & (v_{ij} \text{ fresh}) \end{array}$$

- Converting

$$\begin{array}{lcl} \Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg \alpha = 1 \wedge \neg \beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{array}$$

we obtain:

$$\begin{array}{lcl} \Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg \alpha_1 \wedge \neg \beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i])) \end{array}$$

Step 2: Convert to Boolean Constraints

- Convert the previous constraint system to boolean constraints as follows:

$$\begin{array}{lcll} C_i = C_i & \Leftrightarrow & \text{true} & \\ C_i = C_j & \Leftrightarrow & \text{false} & i \neq j \\ v_i = C_j & \Leftrightarrow & v_{ij} & (v_{ij} \text{ fresh}) \end{array}$$

- Converting

$$\begin{array}{lcl} \Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{array}$$

we obtain:

$$\begin{array}{lcl} \Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i])) \end{array}$$

Step 2: Convert to Boolean Constraints

- Convert the previous constraint system to boolean constraints as follows:

$$\begin{array}{lcll} C_i = C_i & \Leftrightarrow & \text{true} & \\ C_i = C_j & \Leftrightarrow & \text{false} & i \neq j \\ v_i = C_j & \Leftrightarrow & v_{ij} & (v_{ij} \text{ fresh}) \end{array}$$

- Converting

$$\begin{array}{lcl} \Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{array}$$

we obtain:

$$\begin{array}{lcl} \Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i])) \end{array}$$

Step 2: Convert to Boolean Constraints

- Convert the previous constraint system to boolean constraints as follows:

$$\begin{array}{lcll} C_i = C_i & \Leftrightarrow & \text{true} & \\ C_i = C_j & \Leftrightarrow & \text{false} & i \neq j \\ v_i = C_j & \Leftrightarrow & v_{ij} & (v_{ij} \text{ fresh}) \end{array}$$

- Converting

$$\begin{array}{rcl} \Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{array}$$

we obtain:

$$\begin{array}{rcl} \Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i])) \end{array}$$

Step 2: Convert to Boolean Constraints

- Since each variable v_i must have exactly one abstract value C_j, the boolean constraints must satisfy the following additional existence and uniqueness constraints:

1. Uniqueness : $\psi_{\text{unique}} = (\bigwedge_{j \neq k} \neg(v_{ij} \wedge v_{ik}))$
2. Existence : $\psi_{\text{exist}} = (\bigvee_{j} v_{ij})$

Step 2: Convert to Boolean Constraints

- Since each variable v_i must have exactly one abstract value C_j, the boolean constraints must satisfy the following additional existence and uniqueness constraints:

  1. Uniqueness : $\psi_{\text{unique}} = (\bigwedge_{j \neq k} \neg(v_{ij} \wedge v_{ik}))$
  2. Existence : $\psi_{\text{exist}} = (\bigvee_j v_{ij})$

- To enforce these additional existence and uniqueness constraints, define satisfiability and validity as follows:

$$\mathrm{SAT}^*(\phi) \equiv \mathrm{SAT}(\phi \wedge \psi_{\text{exist}} \wedge \psi_{\text{unique}})$$
$$\mathrm{VALID}^*(\phi) \equiv (\{\psi_{\text{exist}}\} \cup \{\psi_{\text{unique}}\} \models \phi)$$

Step 2: Convert to Boolean Constraints

- Since each variable v_i must have exactly one abstract value C_j, the boolean constraints must satisfy the following additional existence and uniqueness constraints:

$$\begin{array}{lll} 1. & \text{Uniqueness}: & \psi_{\text{unique}} = (\bigwedge_{j \neq k} \neg(v_{ij} \wedge v_{ik})) \\ 2. & \text{Existence}: & \psi_{\text{exist}} = (\bigvee_j v_{ij}) \end{array}$$

- To enforce these additional existence and uniqueness constraints, define satisfiability and validity as follows:

$$\begin{array}{l} \text{SAT}^*(\phi) \equiv \text{SAT}(\phi \wedge \psi_{\text{exist}} \wedge \psi_{\text{unique}}) \\ \text{VALID}^*(\phi) \equiv (\{\psi_{\text{exist}}\} \cup \{\psi_{\text{unique}}\} \models \phi) \end{array}$$

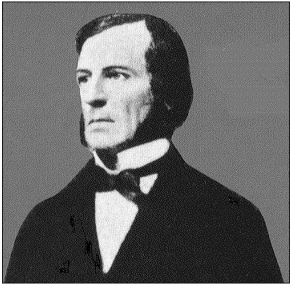
- For instance, using the variables in the previous example,

$$\begin{array}{l} \text{UNSAT}^*(\alpha_1 \wedge \alpha_2) \\ \text{VALID}^*(\beta_1 \vee \beta_2 \vee \beta_3) \end{array}$$

Step 3: Eliminate Choice Variables

$$\mathsf{SNC}(\phi, \beta) = \phi[\mathsf{true}/\beta] \vee \phi[\mathsf{false}/\beta]$$

$$\mathsf{WSC}(\phi, \beta) = \phi[\mathsf{true}/\beta] \wedge \phi[\mathsf{false}/\beta]$$



Resulting Constraints

$$E = \begin{bmatrix} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & [\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & [\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])] \end{bmatrix}$$

Resulting Constraints

$$E_{\mathsf{NC}} = \begin{bmatrix} \lceil \Pi_{f_1,\alpha,C_1} \rceil & = & \phi'_{11}(\vec{\alpha_1}, \lceil \vec{\Pi} \rceil [\vec{b}_1/\vec{\alpha}]) \\ & \vdots & \\ \lceil \Pi_{f_k,\alpha,C_n} \rceil & = & \phi'_{kn}(\vec{\alpha}_k, \lceil \vec{\Pi} \rceil [\vec{b}_k/\vec{\alpha}]) \end{bmatrix}$$

$$E_{\mathsf{SC}} = \begin{bmatrix} \lfloor \Pi_{f_1,\alpha,C_1} \rfloor & = & \phi'_{11}(\vec{\alpha_1}, \lfloor \vec{\Pi} \rfloor [\vec{b}_1/\vec{\alpha}]) \\ & \vdots & \\ \lfloor \Pi_{f_k,\alpha,C_n} \rfloor & = & \phi'_{kn}(\vec{\alpha}_k, \lfloor \vec{\Pi} \rfloor [\vec{b}_k/\vec{\alpha}]) \end{bmatrix}$$

No more choice variables

Resulting Constraints

$$E_{\mathsf{NC}} = \begin{bmatrix} \lceil \Pi_{f_1,\alpha,C_1} \rceil & = & \phi'_{11}(\vec{\alpha_1}, \lceil \vec{\Pi} \rceil [\vec{b}_1/\vec{\alpha}]) \\ & \vdots & \\ \lceil \Pi_{f_k,\alpha,C_n} \rceil & = & \phi'_{kn}(\vec{\alpha}_k, \lceil \vec{\Pi} \rceil [\vec{b}_k/\vec{\alpha}]) \end{bmatrix}$$

$$E_{\mathsf{SC}} = \begin{bmatrix} \lfloor \Pi_{f_1,\alpha,C_1} \rfloor & = & \phi'_{11}(\vec{\alpha_1}, \lfloor \vec{\Pi} \rfloor [\vec{b}_1/\vec{\alpha}]) \\ & \vdots & \\ \lfloor \Pi_{f_k,\alpha,C_n} \rfloor & = & \phi'_{kn}(\vec{\alpha}_k, \lfloor \vec{\Pi} \rfloor [\vec{b}_k/\vec{\alpha}]) \end{bmatrix}$$

But still recursive

Example

- If we eliminate the choice variables from the constraint

$$\begin{aligned}\Pi_{f,\alpha,C_1} \quad = \quad & (\alpha_1 \vee \beta_2 \vee \\ & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_{\text{i}}'/\beta_{\text{i}}]))\end{aligned}$$

  we obtain:

Example

- If we eliminate the choice variables from the constraint

$$\begin{array}{rcl} \Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\mathrm{true}/\alpha_1][\mathrm{false}/\alpha_2][\mathrm{false}/\alpha_3][\beta_{\mathrm{i}}'/\beta_{\mathrm{i}}])) \end{array}$$

we obtain:

$$\begin{array}{rcl} \lceil \Pi_{f,\alpha,C_1} \rceil & = & (\alpha_1 \vee \mathrm{true} \vee \\ & & ((\neg\alpha_1 \wedge \neg\mathrm{true} \wedge \Pi_{\mathrm{f},\alpha,\mathrm{C}_1}[\mathrm{true}/\alpha_1][\mathrm{false}/\alpha_2][\mathrm{false}/\alpha_3])) \\ & & \quad\quad \vee \\ & & (\alpha_1 \vee \mathrm{false} \vee \\ & & ((\neg\alpha_1 \wedge \neg\mathrm{false} \wedge \Pi_{\mathrm{f},\alpha,\mathrm{C}_1}[\mathrm{true}/\alpha_1][\mathrm{false}/\alpha_2][\mathrm{false}/\alpha_3])) \end{array}$$

Example

- If we eliminate the choice variables from the constraint

$$\begin{array}{rcl}\Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_{\text{i}}'/\beta_{\text{i}}]))\end{array}$$

  we obtain:

$$\begin{array}{rcl}\lceil \Pi_{f,\alpha,C_1} \rceil & = & \text{true} \quad \vee \\ & & (\alpha_1 \vee \text{false} \vee \\ & & ((\neg\alpha_1 \wedge \neg\text{false} \wedge \Pi_{\text{f},\alpha,\text{C}_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3]))\end{array}$$

Example

- If we eliminate the choice variables from the constraint

$$\begin{array}{rcl}\Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\mathrm{true}/\alpha_1][\mathrm{false}/\alpha_2][\mathrm{false}/\alpha_3][\beta_{\mathrm{i}}'/\beta_{\mathrm{i}}]))\end{array}$$

  we obtain:

$$\lceil \Pi_{f,\alpha,C_1} \rceil \quad = \quad \mathrm{true}$$

Example

- If we eliminate the choice variables from the constraint

$$\begin{aligned}\Pi_{f,\alpha,C_1} \quad = \quad & (\alpha_1 \vee \beta_2 \vee \\ & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\mathrm{true}/\alpha_1][\mathrm{false}/\alpha_2][\mathrm{false}/\alpha_3][\beta_\mathrm{i}'/\beta_\mathrm{i}]))\end{aligned}$$

we obtain:

$$\begin{aligned}\lceil \Pi_{f,\alpha,C_1} \rceil \quad = \quad & \mathrm{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor \quad = \quad & (\alpha_1 \vee \mathrm{true} \vee \\ & ((\neg\alpha_1 \wedge \neg\mathrm{true} \wedge \Pi_{\mathrm{f},\alpha,\mathrm{C}_1}[\mathrm{true}/\alpha_1][\mathrm{false}/\alpha_2][\mathrm{false}/\alpha_3])) \\ & \qquad \wedge \\ & (\alpha_1 \vee \mathrm{false} \vee \\ & ((\neg\alpha_1 \wedge \neg\mathrm{false} \wedge \Pi_{\mathrm{f},\alpha,\mathrm{C}_1}[\mathrm{true}/\alpha_1][\mathrm{false}/\alpha_2][\mathrm{false}/\alpha_3]))\end{aligned}$$

Example

- If we eliminate the choice variables from the constraint

$$\begin{aligned}\Pi_{f,\alpha,C_1} \quad = \quad & (\alpha_1 \vee \beta_2 \vee \\ & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_{\mathrm{i}}'/\beta_{\mathrm{i}}]))\end{aligned}$$

we obtain:

$$\begin{aligned}\lceil \Pi_{f,\alpha,C_1} \rceil \quad & = \quad \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor \quad & = \quad \text{true} \quad \wedge \\ & \quad\ (\alpha_1 \vee (\neg\alpha_1 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3]))\end{aligned}$$

Example

- If we eliminate the choice variables from the constraint

$$\begin{array}{rcl} \Pi_{f,\alpha,C_1} & = & (\alpha_1 \vee \beta_2 \vee \\ & & ((\neg\alpha_1 \wedge \neg\beta_2 \wedge \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_\text{i}'/\beta_\text{i}])) \end{array}$$

we obtain:

$$\begin{array}{rcl} \lceil \Pi_{f,\alpha,C_1} \rceil & = & \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor & = & \alpha_1 \vee \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3] \end{array}$$

Step 4: Preservation of SNC's and WSC's under Syntactic Substitution

- For subsequent fixed-point computation, the constraints must preserve SNC's and WSC's under syntactic substitution.

Step 4: Preservation of SNC's and WSC's under Syntactic Substitution

- For subsequent fixed-point computation, the constraints must preserve SNC's and WSC's under syntactic substitution.
- In their current form, E_{NC} and E_{SC} do not have this property for two reasons:

Step 4: Preservation of SNC's and WSC's under Syntactic Substitution

- For subsequent fixed-point computation, the constraints must preserve SNC's and WSC's under syntactic substitution.

- In their current form, E_{NC} and E_{SC} do not have this property for two reasons:

  - Recall from earlier: $\neg\lceil\phi\rceil \not\Leftrightarrow \lceil\neg\phi\rceil$ and $\neg\lfloor\phi\rfloor \not\Leftrightarrow \lfloor\neg\phi\rfloor$

Step 4: Preservation of SNC's and WSC's under Syntactic Substitution

- For subsequent fixed-point computation, the constraints must preserve SNC's and WSC's under syntactic substitution.
- In their current form, E_{NC} and E_{SC} do not have this property for two reasons:
  - Recall from earlier: $\neg\lceil\phi\rceil \not\Leftrightarrow \lceil\neg\phi\rceil$ and $\neg\lfloor\phi\rfloor \not\Leftrightarrow \lfloor\neg\phi\rfloor$
  - Contradictions and tautologies must be explicitly enforced when applying substitution
    - Consider $\Pi_{f,\alpha,C_1} \wedge \Pi_{f,\alpha,C_2}$ where $\lceil\Pi_{f,\alpha,C_1}\rceil$ and $\lceil\Pi_{f,\alpha,C_2}\rceil$ are both true

Step 4: Preservation under Syntactic Substitution I

- To deal with the first problem:

Step 4: Preservation under Syntactic Substitution I

- To deal with the first problem:
  - Either replace $\neg\Pi_{f,\alpha,c_i}$ with $\bigvee_{j\neq i}\Pi_{f,\alpha,c_j}$

Step 4: Preservation under Syntactic Substitution I

- To deal with the first problem:
  - Either replace $\neg\Pi_{f,\alpha,c_i}$ with $\bigvee_{j\neq i}\Pi_{f,\alpha,c_j}$
  - Or use the property $\lceil\neg\phi\rceil \Leftrightarrow \neg\lfloor\phi\rfloor$ and $\lfloor\neg\phi\rfloor \Leftrightarrow \neg\lceil\phi\rceil$

Step 4: Preservation under Syntactic Substitution I

- To deal with the first problem:
  - Either replace $\neg\Pi_{f,\alpha,c_i}$ with $\bigvee_{j\neq i}\Pi_{f,\alpha,c_j}$
  - Or use the property $\lceil\neg\phi\rceil \Leftrightarrow \neg\lfloor\phi\rfloor$ and $\lfloor\neg\phi\rfloor \Leftrightarrow \neg\lceil\phi\rceil$
- The latter requires simultaneous fixpoint computation of strongest necessary and weakest sufficient conditions

Step 4: Preservation under Syntactic Substitution I

- To deal with the first problem:
  - Either replace $\neg\Pi_{f,\alpha,c_i}$ with $\bigvee_{j\neq i}\Pi_{f,\alpha,c_j}$
  - Or use the property $\lceil\neg\phi\rceil \Leftrightarrow \neg\lfloor\phi\rfloor$ and $\lfloor\neg\phi\rfloor \Leftrightarrow \neg\lceil\phi\rceil$
- The latter requires simultaneous fixpoint computation of strongest necessary and weakest sufficient conditions
- But important for a practical implementation

Step 4: Preservation under Syntactic Substitution II

- A simple way to enforce contradictions (for necessary conditions) and tautologies (for sufficient conditions):

Step 4: Preservation under Syntactic Substitution II

- A simple way to enforce contradictions (for necessary conditions) and tautologies (for sufficient conditions):
  - For Necessary Conditions: Convert to DNF and drop contradictions of the form $\Pi_{f,\alpha,C_i} \wedge \Pi_{f,\alpha,C_j}$ and $\Pi_{f,\alpha,C_i} \wedge \neg\Pi_{f,\alpha,C_i}$ in each clause

Step 4: Preservation under Syntactic Substitution II

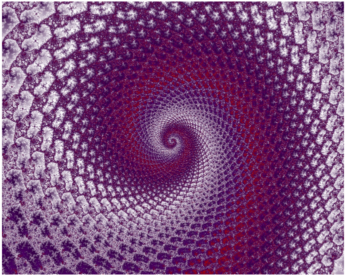
- A simple way to enforce contradictions (for necessary conditions) and tautologies (for sufficient conditions):
  - For Necessary Conditions: Convert to DNF and drop contradictions of the form $\Pi_{f,\alpha,C_i} \wedge \Pi_{f,\alpha,C_j}$ and $\Pi_{f,\alpha,C_i} \wedge \neg\Pi_{f,\alpha,C_i}$ in each clause
  - For Sufficient Conditions: Convert to CNF and drop tautologies

Step 4: Preservation under Syntactic Substitution II

- A simple way to enforce contradictions (for necessary conditions) and tautologies (for sufficient conditions):
  - For Necessary Conditions: Convert to DNF and drop contradictions of the form $\Pi_{f,\alpha,C_i} \wedge \Pi_{f,\alpha,C_j}$ and $\Pi_{f,\alpha,C_i} \wedge \neg\Pi_{f,\alpha,C_i}$ in each clause
  - For Sufficient Conditions: Convert to CNF and drop tautologies

The resulting constraints preserve strongest necessary and weakest sufficient conditions under syntactic substitution.

Step 5: Compute fixed point



Since the modified system of constraints preserve strongest necessary and weakest sufficient conditions under syntactic substitution, compute a fixed-point solution by repeated substitution

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Original constraint:

$$\begin{array}{rcl} \Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{array}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Original constraint:

$$\begin{aligned}\Pi_{f,\alpha,C_1} &= (\alpha = 1 \vee \beta = 2 \vee \\ &\quad ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))\end{aligned}$$

In the previous step, we computed:

$$\begin{aligned}\lceil \Pi_{f,\alpha,C_1} \rceil &= \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor &= \alpha_1 \vee \lfloor \Pi_{f,\alpha,C_1} \rfloor [\text{true}/\alpha_1]\end{aligned}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Original constraint:

$$\begin{array}{rcl} \Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{array}$$

Compute greatest fixed-point:

$$\begin{array}{rcl} \lceil \Pi_{f,\alpha,C_1} \rceil & = & \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor & = & \alpha_1 \vee \text{false} = \alpha_1 \end{array}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Original constraint:

$$\begin{array}{rcl}\Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))\end{array}$$

Compute greatest fixed-point:

$$\begin{array}{rcl}\lceil \Pi_{f,\alpha,C_1} \rceil & = & \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor & = & \alpha_1 \vee \alpha_1[\text{true}/\alpha_1]\end{array}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Original constraint:

$$\begin{array}{rcl}\Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))\end{array}$$

Compute greatest fixed-point:

$$\begin{array}{rcl}\lceil \Pi_{f,\alpha,C_1} \rceil & = & \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor & = & \alpha_1 \vee \text{true}\end{array}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Original constraint:

$$\begin{array}{rcl} \Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \end{array}$$

Compute greatest fixed-point:

$$\begin{array}{rcl} \lceil \Pi_{f,\alpha,C_1} \rceil & = & \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor & = & \text{true} \end{array}$$

Example

```
int f(int x) {
   int y = getUserInput();
   if(x == 1 || y == 2) return 1;
   return f(1);
}
```

Original constraint:

$$\begin{array}{rcl}\Pi_{f,\alpha,C_1} & = & (\alpha = 1 \vee \beta = 2 \vee \\ & & ((\neg\alpha = 1 \wedge \neg\beta = 2 \wedge \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))\end{array}$$

Compute greatest fixed-point:

$$\begin{array}{rcl}\lceil \Pi_{f,\alpha,C_1} \rceil & = & \text{true} \\ \lfloor \Pi_{f,\alpha,C_1} \rfloor & = & \text{true}\end{array}$$

- The sufficient condition expresses that the function MUST return 1 because $\mathrm{VALID}(\lfloor \Pi_{\mathrm{f},\alpha,\mathrm{C}_1} \rfloor)$ holds.

The Main Result

Main Result

- The technique is sound and complete for answering satisfiability and validity queries with respect to some user-provided finite abstraction.

The Main Result

Main Result

- The technique is sound and complete for answering satisfiability and validity queries with respect to some user-provided finite abstraction.
- No choice variables

The Main Result

Main Result

- The technique is sound and complete for answering satisfiability and validity queries with respect to some user-provided finite abstraction.
- No choice variables $\Rightarrow$ Small formulas

The Main Result

Main Result

- The technique is sound and complete for answering satisfiability and validity queries with respect to some user-provided finite abstraction.
- No choice variables $\Rightarrow$ Small formulas $\Rightarrow$ Good scalability

Experiments I

- We compute the full interprocedural constraint -in terms of SNC's and WSC's- for every pointer dereference in OpenSSH, Samba and the Linux kernel ($>$6 MLOC).

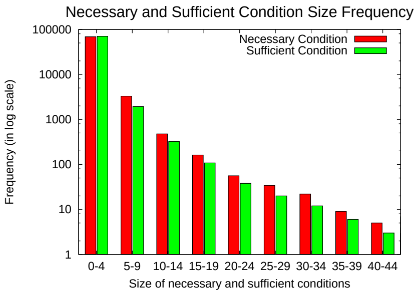
Experiments I

- We compute the full interprocedural constraint -in terms of SNC's and WSC's- for every pointer dereference in OpenSSH, Samba and the Linux kernel (>6 MLOC).
- Stress-test: pointer dereferences are ubiquitous in C programs.

Experiments I

- We compute the full interprocedural constraint -in terms of SNC's and WSC's- for every pointer dereference in OpenSSH, Samba and the Linux kernel (>6 MLOC).

- Stress-test: pointer dereferences are ubiquitous in C programs.



Experiments II

- We also used this technique for an interprocedurally path-sensitive null dereference analysis.

Experiments II

- We also used this technique for an interprocedurally path-sensitive null dereference analysis.

	Interprocedurally Path-sensitive			Intraprocedurally Path-sensitive		
	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1
Total Reports	3	48	171	21	379	1495
Bugs	1	17	134	1	17	134
False Positives	2	25	37	20	356	1344
Undecided	0	6	17	0	6	17
Report to Bug Ratio	3	2.8	1.3	21	22.3	11.2

Experiments II

- We also used this technique for an interprocedurally path-sensitive null dereference analysis.

	Interprocedurally Path-sensitive			Intraprocedurally Path-sensitive		
	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1
Total Reports	3	48	171	21	379	1495
Bugs	1	17	134	1	17	134
False Positives	2	25	37	20	356	1344
Undecided	0	6	17	0	6	17
Report to Bug Ratio	3	2.8	1.3	21	22.3	11.2

Experiments II

- We also used this technique for an interprocedurally path-sensitive null dereference analysis.

	Interprocedurally Path-sensitive			Intraprocedurally Path-sensitive		
	OpenSSH 4.3p2	**Samba 3.0.23b**	**Linux 2.6.17.1**	**OpenSSH 4.3p2**	**Samba 3.0.23b**	**Linux 2.6.17.1**
Total Reports	3	48	171	21	379	1495
Bugs	1	17	134	1	17	134
False Positives	2	25	37	20	356	1344
Undecided	0	6	17	0	6	17
Report to Bug Ratio	3	2.8	1.3	21	22.3	11.2

Experiments II

- We also used this technique for an interprocedurally path-sensitive null dereference analysis.

	Interprocedurally Path-sensitive			Intraprocedurally Path-sensitive		
	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1	OpenSSH 4.3p2	Samba 3.0.23b	Linux 2.6.17.1
Total Reports	3	48	171	21	379	1495
Bugs	1	17	134	1	17	134
False Positives	2	25	37	20	356	1344
Undecided	0	6	17	0	6	17
Report to Bug Ratio	3	2.8	1.3	21	22.3	11.2

Experiments II

- We also used this technique for an interprocedurally path-sensitive null dereference analysis.

	Interprocedurally Path-sensitive			Intraprocedurally Path-sensitive		
	OpenSSH 4.3p2	**Samba 3.0.23b**	**Linux 2.6.17.1**	**OpenSSH 4.3p2**	**Samba 3.0.23b**	**Linux 2.6.17.1**
Total Reports	3	48	171	21	379	1495
Bugs	1	17	134	1	17	134
False Positives	2	25	37	20	356	1344
Undecided	0	6	17	0	6	17
Report to Bug Ratio	3	2.8	1.3	21	22.3	11.2

- Observed close to an order of magnitude reduction of false positives without resorting to (potentially unsound) ad-hoc heuristics.

Future Directions I

- Caveat: Previous experiments do not track NULL values in unbounded data structures.

Future Directions I

- Caveat: Previous experiments do not track NULL values in unbounded data structures.
- Underlying framework collapses all unbounded data structures into one summary node

Future Directions I

- Caveat: Previous experiments do not track NULL values in unbounded data structures.
- Underlying framework collapses all unbounded data structures into one summary node
- Imprecise for verifying memory safety.

Future Directions I

- Caveat: Previous experiments do not track NULL values in unbounded data structures.
- Underlying framework collapses all unbounded data structures into one summary node
- Imprecise for verifying memory safety.
- Analysis of contents of position dependent data structures, such as arrays, linked lists etc., is one of our current projects.

Future Directions II

- Computing strongest necessary and weakest sufficient conditions in richer theories

Future Directions II

- Computing strongest necessary and weakest sufficient conditions in richer theories
  - e.g., theory of uninterpreted functions; combined theory of linear arithmetic over integers and uninterpreted functions, . . .

Future Directions II

- Computing strongest necessary and weakest sufficient conditions in richer theories
  - e.g., theory of uninterpreted functions; combined theory of linear arithmetic over integers and uninterpreted functions, ...
  - Closely related to *cover algorithms* for existential quantifier elimination ("Cover Algorithms and Their Combination" by Gulwani and Musuvathi)

Related Work

T. Ball and S. Rajamani.
Bebop: A symbolic model checker for boolean programs.
In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130, London, UK, 2000. Springer-Verlag.

M. Das, S. Lerner, and M. Seigle.
ESP: Path-sensitive program verification in polynomial time.
In *Proc. Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan.
Abstractions from proofs.
In *Proc. 31st Symposium on Principles of Programming Languages*, pages 232–244, 2004.

A. Mycroft.
Polymorphic type schemes and recursive definitions.
In *Proc. Colloquium on International Symposium on Programming*, pages 217–228, 1984.

T. Reps, S. Horwitz, and M. Sagiv.
Precise interprocedural dataflow analysis via graph reachability.
In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.

D. Schmidt.
A calculus of logical relations for over- and underapproximating static analyses.
Science of Computer Programming, 64(1):29–53, 2007.

Y. Xie and A. Aiken.
Scalable error detection using boolean satisfiability.
SIGPLAN Not., 40(1):351–363, 2005.

Thank You For Listening!