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

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

- User Input
- Network data

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

- User Input
- Network data
- System state

```c
int* p = malloc(sizeof(int)*num elems);
if(p == NULL) exit(1);
```

- Many more, e.g., calling an unknown function, thread scheduling, . . .
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
In contrast to uncertainty, imprecision arises from the abstraction intentionally chosen by the analysis designer.
In contrast to uncertainty, imprecision arises from the abstraction intentionally chosen by the analysis designer.

But imprecision results in similar consequences as uncertainty.
Many program analysis systems do not reason about unbounded data structures or abstract data types.

```java
int elem = array[i];
assert(elem != -1);
```
Constraint-Based Analysis in the Presence of Uncertainty and Imprecision

Imprecision

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

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

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

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

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

```c
if(x >= MAX)
    return -1;
```
Many program analysis 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.

```c
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);
```
Many program analysis 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

```c
int compute_gcd(int a, int b) {
    // Unimplemented
}

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

- Many program analysis 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

```c
int compute_gcd(int a, int b)
{
    // Question mark indicates a gap in the code
}
assert(x% ? == 0);
```
Imprecision

- Many program analysis 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.
In constraint-based systems, environment choice is typically modeled using unconstrained variables that we call choice variables.
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 $\beta$. 
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 $\beta$.

$\text{SAT}(\beta = 'y')$?
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 $\beta$.

$\text{SAT}(\beta = 'y') ?$ Of course!
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 $\beta$.

\[
\text{SAT}(\beta \neq 'y')?
\]
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 $\beta$.

\[ \text{SAT}(\beta \neq 'y') \text{? Of course!} \]
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 $\beta$.

\[
\text{VALID}(\beta = 'y') \ ?
\]
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 $\beta$.

$$\text{VALID}(\beta = 'y') \ ? \text{ Of course not!}$$
Unfortunately, the use of choice variables may introduce two problems:
Unfortunately, the use of choice variables may introduce two problems:

- **Theoretical**: It is not clear how to solve recursive constraints containing choice variables.
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

```c
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

```c
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

```c
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 $\alpha$, what is the constraint $\Pi_{\alpha,\text{true}}$ under which queryUser returns true?
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);
}

$\Pi_{\alpha, \text{true}} =$?
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);
}

\[ \Pi_{\alpha, \text{true}} = (\alpha = \text{true}) \land ? \]
Recursive Constraint Example

```c
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);
}
```

\[
\Pi_{\alpha,\text{true}} = ((\alpha = \text{true}) \land (\beta = 'y' \lor ?))
\]
Recursive Constraint Example

```c
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\n");
    return queryUser(featureEnabled);
}
```

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

```cpp
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);
}
```

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

```cpp
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);
}
```

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

```cpp
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);
}
```

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

```c
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);
}
```

\[
\Pi_{\alpha, \text{true}} = ((\alpha = \text{true}) \land (\beta = 'y' \lor (\beta \neq 'n' \land \Pi_{\alpha, \text{true}}[\text{true/}\alpha][\beta'/\beta])))
\]
If we solve this constraint naively using standard fix-point computation, we get:

\[ \Pi_{\alpha,\text{true}} = (\alpha = \text{true}) \land (\beta = 'y' \lor \neg (\beta = 'n') \land \Pi_{\alpha,\text{true}}[\text{true}/\alpha][\beta'/\beta])] \]
Recursive Constraint Example

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

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

\[ \text{where } \beta', \text{ and } \beta'' \text{ are choice variables.} \]
If we solve this constraint naively using standard fix-point computation, we get:

\[ \Pi_{\alpha,\text{true}} = (\alpha = \text{true}) \land (\beta = 'y' \lor \neg (\beta = 'n')) \land \\
(\text{true} = \text{true}) \land (\beta' = 'y' \lor \neg (\beta' = 'n')) \land \\
(\text{true} = \text{true}) \land (\beta'' = 'y' \lor \neg (\beta'' = 'n')) \land \\
\ldots \]
Recursive Constraint Example

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

\[
\Pi_{\alpha,\text{true}} = (\alpha = \text{true}) \land (\beta = 'y' \lor \neg(\beta = 'n'))) \land \\
(\text{true} = \text{true}) \land (\beta' = 'y' \lor \neg(\beta' = 'n'))) \land \\
(\text{true} = \text{true}) \land (\beta'' = 'y' \lor \neg(\beta'' = 'n'))) \land \\
\ldots
\]

- 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

```c
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;
}
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 $\alpha$, let us slice the relevant part of the function:
Denoting the argument of `key_new_private` by $\alpha$, let us slice the relevant part of the function:

```c
key_new_private(\alpha) {
    if (\alpha == 1 || \alpha == 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 (\alpha == 3)
        if (BN_new() == NULL) /* fail */
    /* success */
}
```
Scalability Problem Example

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

```c
key_new_private(\alpha) {
    if (\alpha == 1 || \alpha == 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 (\alpha == 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 \( \alpha \), let us slice the relevant part of the function:

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

- We replace each call to `BN_NEW` with a fresh choice variable \( \beta_i \).
Scalability Problem Example

```c
key_new_private(\alpha) {
    if (\alpha == 1 || \alpha == 2 ) {
        if (\beta_1 == 0 || \beta_2 == 0 || \beta_3 == 0 || \beta_4 == 0 || \beta_5 == 0) /* fail */
    }
    if (\alpha == 3)
        if (\beta_6 == 0) /* fail */
    /* success */
}
```

The condition under which the function succeeds is:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]
Scalability Problem Example

key_new_private(α) {
    if (α == 1 || α == 2) {
        if (β_1 == 0 || β_2 == 0 || β_3 == 0 || β_4 == 0 || β_5 == 0) /* fail */
    } else if (α == 3) {
        if (β_6 == 0) /* fail */
    } /* success */
}

■ The condition under which the function succeeds is:

\[(1 \leq α \leq 2 \land (β_1 \neq 0 \land β_2 \neq 0 \land β_3 \neq 0 \land β_4 \neq 0 \land β_5 \neq 0) \lor (α = 3 \land β_6 \neq 0) \lor α \leq 0 \lor α \geq 4)\]
Scalability Problem Example

```c
key_new_private(α) {
    if (α == 1 || α == 2) {
        if (β_1 == 0 || β_2 == 0 || β_3 == 0 || β_4 == 0 || β_5 == 0)
            /* fail */
    }
    if (α == 3)
        if (β_6 == 0) /* fail */
            /* success */
}
```

- The condition under which the function succeeds is:

\[
(1 ≤ α ≤ 2 \land (β_1 ≠ 0 \land β_2 ≠ 0 \land β_3 ≠ 0 \land β_4 ≠ 0 \land β_5 ≠ 0) \\
\lor (α = 3 \land β_6 ≠ 0) \lor α ≤ 0 \lor α ≥ 4)
\]

- Very verbose way of stating the success condition!
Scalability Problem Example

- Now consider some calling context of this function:

```c
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:

```c
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*/?

\[
(1 \leq 1 \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \\
\lor (1 = 3 \land \beta_6 \neq 0) \lor 1 \leq 0 \lor 1 \geq 4) \land \\
(1 \leq 2 \leq 2 \land (\beta'_1 \neq 0 \land \beta'_2 \neq 0 \land \beta'_3 \neq 0 \land \beta'_4 \neq 0 \land \beta'_5 \neq 0) \\
\lor (2 = 3 \land \beta'_6 \neq 0) \lor 2 \leq 0 \lor 2 \geq 4) \land \\
(1 \leq 3 \leq 2 \land (\beta''_1 \neq 0 \land \beta''_2 \neq 0 \land \beta''_3 \neq 0 \land \beta''_4 \neq 0 \land \beta''_5 \neq 0) \\
\lor (3 = 3 \land \beta''_6 \neq 0) \lor 3 \leq 0 \lor 3 \geq 4)
\]
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*/?

\[
(1 \leq 1 \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \\
\lor (1 = 3 \land \beta_6 \neq 0) \lor 1 \leq 0 \lor 1 \geq 4) \land \\
(1 \leq 2 \leq 2 \land (\beta'_1 \neq 0 \land \beta'_2 \neq 0 \land \beta'_3 \neq 0 \land \beta'_4 \neq 0 \land \beta'_5 \neq 0) \\
\lor (2 = 3 \land \beta'_6 \neq 0) \lor 2 \leq 0 \lor 2 \geq 4) \land \\
(1 \leq 3 \leq 2 \land (\beta''_1 \neq 0 \land \beta''_2 \neq 0 \land \beta''_3 \neq 0 \land \beta''_4 \neq 0 \land \beta''_5 \neq 0) \\
\lor (3 = 3 \land \beta''_6 \neq 0) \lor 3 \leq 0 \lor 3 \geq 4)
\]
Conclusion from the Examples

- Introducing choice variables causes problems both with scalability and solving recursive constraints.
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.
An over-approximation $\lceil \phi \rceil$ of a constraint $\phi$ not containing choice variables is implied by the original constraint, i.e. $\lceil \phi \rceil$ is a necessary condition.

$$\phi \Rightarrow \lceil \phi \rceil$$
An over-approximation $\lceil \phi \rceil$ of a constraint $\phi$ 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:

$$\forall \phi'. ((\phi \Rightarrow \phi') \Rightarrow (\lceil \phi \rceil \Rightarrow \phi'))$$
An over-approximation $[\phi]$ of a constraint $\phi$ not containing choice variables is implied by the original constraint, i.e. $[\phi]$ is a necessary condition.

$\phi \Rightarrow [\phi]$

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

$\forall \phi'. ((\phi \Rightarrow \phi') \Rightarrow ([\phi] \Rightarrow \phi'))$

Because strongest necessary condition $[\phi]$ preserves the satisfiability of $\phi$:

$\text{SAT}(\phi) \iff \text{SAT}([\phi])$
SNC Example 1

Consider the constraint from `key_new_private`:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]
SNC Example 1

Consider the constraint from `key_new_private`:

\[
(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \\
\lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)
\]

The strongest necessary condition for this formula is just `true`. 
SNC Example 1

```c
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.
Consider the constraint from the `queryUser` function:

\[
\Pi_{\alpha,\text{true}} = ((\alpha = \text{true}) \land (\beta = 'y' \lor (\beta \neq 'n' \land \Pi_{\alpha,\text{true}}[\text{true}/\alpha][\beta'/\beta])))
\]
SNC Example 2

- Consider the constraint from the `queryUser` function:

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

- The strongest necessary condition for \(\Pi_{\alpha,\text{true}}\) is \(\alpha = \text{true}\).
SNC Example 2

```c
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...

\[ [\neg \phi] \not\iff [\neg \phi] \]
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...

\[ \lnot \phi \not\leftrightarrow \lnot [\phi] \]

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

The weakest sufficient condition $\lceil \phi \rceil$ of formula $\phi$ not containing any choice variables satisfies:

1. $\lceil \phi \rceil \Rightarrow \phi$
2. $\forall \phi'. ((\phi' \Rightarrow \phi) \Rightarrow (\phi' \Rightarrow \lceil \phi \rceil))$
Weakest Sufficient Conditions

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

  (1) $\lfloor \phi \rfloor \Rightarrow \phi$
  (2) $\forall \phi'.((\phi' \Rightarrow \phi) \Rightarrow (\phi' \Rightarrow \lfloor \phi \rfloor))$

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

  $\text{VALID}(\phi) \Leftrightarrow \text{VALID}(\lfloor \phi \rfloor)$
WSC Example 1

Consider the constraint from `key new private`:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]
WSC Example 1

- Consider the constraint from `key_new_private`:
  
  $$
  (1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \\
  \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)
  $$

- The weakest sufficient condition for this formula is
  \( \alpha \leq 0 \lor \alpha \geq 4. \)
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:

\[ \Pi_{\alpha, \text{true}} = ((\alpha = \text{true}) \land (\beta = 'y' \lor (\beta \neq 'n' \land \Pi_{\alpha, \text{true}[\text{true}/\alpha][\beta'/\beta]}))) \]
WSC Example 2

- Consider the constraint from the `queryUser` function:

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

- The weakest sufficient condition for this formula is false.
WSC Example 2

```c
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.
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 $\phi$’s strongest necessary condition, $\neg \lceil \phi \rceil$. 
Negation Example

- Consider once more the constraint:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]
Negation Example

- Consider once more the constraint:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]

- The strongest necessary and weakest sufficient conditions for success:

\[(true, \alpha \leq 0 \lor \alpha \geq 4)\]
Negation Example

- Consider once more the constraint:
  
  $$(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \\
  \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)$$

- The strongest necessary and weakest sufficient conditions for success:
  
  $$(\text{true, } \alpha \leq 0 \lor \alpha \geq 4)$$

- Strongest necessary and weakest sufficient conditions for failure:
  
  $$(?, ?)$$
Negation Example

- Consider once more the constraint:

\[
(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \\
\lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)
\]

- The strongest necessary and weakest sufficient conditions for success:

\[(true, \alpha \leq 0 \lor \alpha \geq 4)\]

- Strongest necessary and weakest sufficient conditions for failure:

\[(\neg (\alpha \leq 0 \lor \alpha \geq 4), ?)\]
Negation Example

- Consider once more the constraint:

\[
(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \\
\lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)
\]

- The strongest necessary and weakest sufficient conditions for success:

\[(\text{true}, \alpha \leq 0 \lor \alpha \geq 4)\]

- Strongest necessary and weakest sufficient conditions for failure:

\[(1 \leq \alpha \leq 3, ?)\]
Negation Example

- Consider once more the constraint:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]

- The strongest necessary and weakest sufficient conditions for success:

\[(\text{true}, \alpha \leq 0 \lor \alpha \geq 4)\]

- Strongest necessary and weakest sufficient conditions for failure:

\[(1 \leq \alpha \leq 3, \neg \text{true})\]
Negation Example

- Consider once more the constraint:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]

- The strongest necessary and weakest sufficient conditions for success:

\[(true, \alpha \leq 0 \lor \alpha \geq 4)\]

- Strongest necessary and weakest sufficient conditions for failure:

\[(1 \leq \alpha \leq 3, false)\]
Negation Example

Consider once more the constraint:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]

The strongest necessary and weakest sufficient conditions for success:

\[(true, \alpha \leq 0 \lor \alpha \geq 4)\]

Strongest necessary and weakest sufficient conditions for failure:

\[(1 \leq \alpha \leq 3, false)\]

Nothing guarantees key_new_private will fail; i.e. weakest sufficient condition is false.
Negation Example

- Consider once more the constraint:

\[(1 \leq \alpha \leq 2 \land (\beta_1 \neq 0 \land \beta_2 \neq 0 \land \beta_3 \neq 0 \land \beta_4 \neq 0 \land \beta_5 \neq 0) \lor (\alpha = 3 \land \beta_6 \neq 0) \lor \alpha \leq 0 \lor \alpha \geq 4)\]

- The strongest necessary and weakest sufficient conditions for success:

\[(\text{true}, \alpha \leq 0 \lor \alpha \geq 4)\]

- Strongest necessary and weakest sufficient conditions for failure:

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

- Requested key must have type \texttt{KEY\_RSA1}, \texttt{KEY\_RSA}, or \texttt{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.
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.
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.
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
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.
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](image)

![Sound & Complete Scale](image)
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.
void process_file(File* f) {
    printf(‘‘Open new file?
’’);
    char user_input = getUserInput();
    if(user_input == 'y')
        f = fopen(NEW_FILE_NAME);
    process_file_internal(f);
    if(user_input == 'y')
        fclose(f);
}
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);
}
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

```c
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
If we are interested in answering **may** and **must** queries, we can safely eliminate choice variables at their scoping boundaries.
Choice Variables and Scope

```c
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

```c
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 \texttt{f} be dereferenced by \texttt{process\_file}? \textbf{YES!}
Choice Variables and Scope

```c
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`?
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);
}
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 $\Pi_{f_i,\alpha,C_j}$ under which a function $f_i$, given input $\alpha$, returns some abstract value $C_j$: 
Step 1: Generate constraints

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

$$E = \begin{bmatrix}
[\Pi_{f_1, \alpha, C_i}] &=& [\phi_{1i}(\bar{\alpha}_1, \bar{\beta}_1, \Pi[\bar{b}_1/\bar{\alpha}][\bar{\beta}'/\bar{\beta}])] \\
\vdots & & \vdots \\
[\Pi_{f_k, \alpha, C_i}] &=& [\phi_{ki}(\bar{\alpha}_k, \bar{\beta}_k, \Pi[\bar{b}_k/\bar{\alpha}][\bar{\beta}'/\bar{\beta}])] 
\end{bmatrix}$$
Step 1: Generate constraints

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

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

- Constraints $\phi_{ij}$ are boolean combinations of $\alpha = C_i$, $\beta = C_i$, $\Pi_{f_i,\alpha,C_j}$ and $C_i = C_j$. 
Step 1: Generate constraints

- Set up a recursive system \( E \) of constraints describing the constraint \( \Pi_{f_i,\alpha,C_j} \) under which a function \( f_i \), given input \( \alpha \), returns some abstract value \( C_j \):

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

- \( \alpha \)'s represent function inputs, provided by the calling context.
Step 1: Generate constraints

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

$$E = \begin{bmatrix}
\Pi_{f_1, \alpha, C_i} &=& \phi_{1i}(\bar{\alpha}_1, \bar{\beta}_1, \Pi[b_1/\bar{\alpha}][\bar{\beta}'/\bar{\beta}]) \\
\vdots & & \vdots \\
\Pi_{f_k, \alpha, C_i} &=& \phi_{ki}(\bar{\alpha}_k, \bar{\beta}_k, \Pi[b_k/\bar{\alpha}][\bar{\beta}'/\bar{\beta}])
\end{bmatrix}$$

- $\beta$'s represent choice variables. The scope of each $\beta$ is the function body in which it is introduced.
Step 1: Generate constraints

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

$$E = \begin{bmatrix}
[\Pi_{f_1,\alpha,C_i}] = [\phi_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \Pi[b_1/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])]

:\vdots

[\Pi_{f_k,\alpha,C_i}] = [\phi_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \Pi[b_k/\vec{\alpha}][\vec{\beta}'/\vec{\beta}])]
\end{bmatrix}$$

- $\Pi$’s on the right hand side result from function calls.
Example

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

```c
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:

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

```c
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:
  
  $C_1 : \{1\}$,  
  $C_2 : \{2\}$,  
  $C_3 : \mathbb{Z} \setminus \{1, 2\}$

- Then,

  \[
  \Pi_{f, \alpha, C_1} = (\alpha = 1 \lor \beta = 2 \lor (\neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f, \alpha, C_1}[1/\alpha][\beta'/\beta]))
  \]
Example

```c
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:

  $C_1 : \{1\}$, $C_2 : \{2\}$, $C_3 : \mathbb{Z} \setminus \{1, 2\}$

- Then,

  $$\Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor \neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f,\alpha,C_1[1/\alpha][\beta'/\beta]})$$
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:

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

- Then,

  $$\Pi_{f, \alpha, C_1} = (\alpha = 1 \lor \beta = 2 \lor ((\neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f, \alpha, C_1}[1/\alpha][\beta'/\beta]))$$
Example

```java
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:
  \[
  C_1 : \{1\}, \quad C_2 : \{2\}, \quad C_3 : \mathbb{Z} \setminus \{1, 2\}
  \]

- Then,
  \[
  \Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor (\neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))
  \]
Step 2: Convert to Boolean Constraints

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

  \[ C_i = C_i \iff \text{true} \]
  \[ C_i = C_j \iff \text{false} \quad i \neq j \]
  \[ v_i = C_j \iff v_{ij} \quad (v_{ij} \text{ fresh}) \]
Step 2: Convert to Boolean Constraints

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

  \[ C_i = C_i \iff \text{true} \]
  \[ C_i = C_j \iff \text{false} \quad i \neq j \]
  \[ v_i = C_j \iff v_{ij} \quad (v_{ij} \text{ fresh}) \]

- Converting

  \[ \Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor \Pi_{f,\alpha,C_1}[\text{true}/\alpha][\beta'/\beta]) \]

we obtain:

\[ \Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor (\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta'_i/\beta_i])) \]
Step 2: Convert to Boolean Constraints

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

  \[ C_i = C_i \iff \text{true} \]
  \[ C_i = C_j \iff \text{false} \quad i \neq j \]
  \[ v_i = C_j \iff v_{ij} \quad (v_{ij} \text{ fresh}) \]

- Converting

  \[ \Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor (\neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f,\alpha,C_1}[true/\alpha][false/\beta])) \]

  we obtain:

  \[ \Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor ((\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[true/\alpha_1][false/\beta_1][false/\beta_2][false/\beta_3][\beta_i/\beta])) \]
Step 2: Convert to Boolean Constraints

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

\[
C_i = C_i \iff \text{true}
\]
\[
C_i = C_j \iff \text{false} \quad i \neq j
\]
\[
v_i = C_j \iff v_{ij} \quad (v_{ij} \text{ fresh})
\]

- Converting

\[
\Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])
\]

we obtain:

\[
\Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor \\
(\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i]))
\]
Step 2: Convert to Boolean Constraints

- Convert the previous constraint system to boolean constraints as follows:
  
  \[ C_i = C_i \iff \text{true} \]
  
  \[ C_i = C_j \iff \text{false} \quad i \neq j \]
  
  \[ v_i = C_j \iff v_{ij} \quad (v_{ij} \text{ fresh}) \]

- Converting

  \[ \Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]) \]

  we obtain:

  \[ \Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i]) \]
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**: $\psi_{\text{unique}} = (\wedge_{j \neq k} \neg(v_{ij} \land v_{ik}))$
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**: $\psi_{\text{unique}} = (\wedge_{j \neq k} \neg (v_{ij} \land v_{ik}))$
  2. **Existence**: $\psi_{\text{exist}} = (\lor_j v_{ij})$

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

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

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

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

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

For instance, using the variables in the previous example,

$$\text{UNSAT}^*(\alpha_1 \land \alpha_2)$$
$$\text{VALID}^*(\beta_1 \lor \beta_2 \lor \beta_3)$$
Step 3: Eliminate Choice Variables

\[
SNC(\phi, \beta) = \phi[true/\beta] \lor \phi[false/\beta]
\]

\[
WSC(\phi, \beta) = \phi[true/\beta] \land \phi[false/\beta]
\]
Resulting Constraints

\[
E = \begin{bmatrix}
[\Pi_{f_1,\alpha,C_i}] &=& [\phi_{1i}(\bar{\alpha}_1, \bar{\beta}_1, \Pi[b_1/\bar{\alpha}][\bar{\beta}'/\bar{\beta}])]

\vdots

[\Pi_{f_k,\alpha,C_i}] &=& [\phi_{ki}(\bar{\alpha}_k, \bar{\beta}_k, \Pi[b_k/\bar{\alpha}][\bar{\beta}'/\bar{\beta}])]
\end{bmatrix}
\]
Resulting Constraints

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

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

No more choice variables
Resulting Constraints

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

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

But still recursive
Example

If we eliminate the choice variables from the constraint

$$
\Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor 
((\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[\text{true/}\alpha_1][\text{false/}\alpha_2][\text{false/}\alpha_3][\beta_i'/\beta_i]))$$

we obtain:
Example

If we eliminate the choice variables from the constraint

\[ \Pi_{f, \alpha, C_1} = (\alpha_1 \lor \beta_2 \lor ((\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f, \alpha, C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3][\beta_i'/\beta_i])) \]

we obtain:

\[ [\Pi_{f, \alpha, C_1}] = (\alpha_1 \lor true \lor ((\neg \alpha_1 \land \neg true \land \Pi_{f, \alpha, C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3]))) \lor (\alpha_1 \lor false \lor ((\neg \alpha_1 \land \neg false \land \Pi_{f, \alpha, C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3]))) \]
Example

If we eliminate the choice variables from the constraint

\[
\Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor ((\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta'_i/\beta_i]))
\]

we obtain:

\[
[\Pi_{f,\alpha,C_1}] = \text{true} \lor (\alpha_1 \lor \text{false} \lor ((\neg \alpha_1 \land \neg \text{false} \land \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3]))
\]
Example

If we eliminate the choice variables from the constraint

\[ \Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor ((\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i])) \]

we obtain:

\[ [\Pi_{f,\alpha,C_1}] = \text{true} \]
Example

- If we eliminate the choice variables from the constraint

$$\Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor ((\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3][\beta'_i/\beta_i]))$$

we obtain:

$$[\Pi_{f,\alpha,C_1}] = true$$
$$[\Pi_{f,\alpha,C_1}] = (\alpha_1 \lor true\lor ((\neg \alpha_1 \land \neg true \land \Pi_{f,\alpha,C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3])) \land (\alpha_1 \lor false\lor ((\neg \alpha_1 \land \neg false \land \Pi_{f,\alpha,C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3]))$$
Example

If we eliminate the choice variables from the constraint

\[ \Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor (\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3][\beta'_i/\beta_i])) \]

we obtain:

\[
\begin{align*}
[\Pi_{f,\alpha,C_1}] & = \text{true} \\
[\Pi_{f,\alpha,C_1}] & = \text{true} \land \\
(\alpha_1 \lor (\neg \alpha_1 \land \Pi_{f,\alpha,C_1}[true/\alpha_1][false/\alpha_2][false/\alpha_3]))
\end{align*}
\]
Example

If we eliminate the choice variables from the constraint

\[ \Pi_{f,\alpha,C_1} = (\alpha_1 \lor \beta_2 \lor (\neg \alpha_1 \land \neg \beta_2 \land \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3][\beta_i'/\beta_i])) \]

we obtain:

\[
\begin{align*}
[\Pi_{f,\alpha,C_1}] & = \text{true} \\
[\Pi_{f,\alpha,C_1}] & = \alpha_1 \lor \Pi_{f,\alpha,C_1}[\text{true}/\alpha_1][\text{false}/\alpha_2][\text{false}/\alpha_3]
\end{align*}
\]
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: \( \neg[\phi] \not\leftrightarrow [\neg\phi] \) and \( \neg[\phi] \not\leftrightarrow [\neg\phi] \)
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: $\neg [\phi] \nleftrightarrow [\neg \phi]$ and $\neg [\phi] \nleftrightarrow [\neg \phi]$
  - Contradictions and tautologies must be explicitly enforced when applying substitution

- Consider $\Pi_{f,\alpha,C_1} \land \Pi_{f,\alpha,C_2}$ where $[\Pi_{f,\alpha,C_1}]$ and $[\Pi_{f,\alpha,C_2}]$ are both true
Step 4: Preservation under Syntactic Substitution I

- To deal with the first problem:

  \[ \neg \Pi_{f, \alpha, c}^i \]

  Or use the property

  \[ \lceil \neg \varphi \rceil \iff \neg \lfloor \varphi \rfloor \]

  \[ \lfloor \neg \varphi \rfloor \iff \neg \lceil \varphi \rceil \]

  The latter requires simultaneous fixpoint computation of the strongest necessary and weakest sufficient conditions.

  But important for a practical implementation.
Step 4: Preservation under Syntactic Substitution I

To deal with the first problem:
- Either replace $\neg \Pi_{f,\alpha,c_i}$ with $\bigvee_{j \neq i} \Pi_{f,\alpha,c_j}$
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 $\lceil \neg \phi \rceil \iff \neg \lfloor \phi \rfloor$ and $\lfloor \neg \phi \rfloor \iff \neg \lceil \phi \rceil$
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
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
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 $\Pi_{f,\alpha,C_i} \land \Pi_{f,\alpha,C_j}$ and $\Pi_{f,\alpha,C_i} \land \neg \Pi_{f,\alpha,C_i}$ 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 $\Pi_{f,\alpha,C_i} \land \Pi_{f,\alpha,C_j}$ and $\Pi_{f,\alpha,C_i} \land \neg \Pi_{f,\alpha,C_i}$ in each clause
  - **For Sufficient Conditions:** Convert to CNF and drop tautologies
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 \( \Pi f,\alpha,C_i \land \Pi f,\alpha,C_j \) and \( \Pi f,\alpha,C_i \land \neg \Pi f,\alpha,C_i \) 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.
int f(int x) {
    int y = getUserInput();
    if (x == 1 || y == 2) return 1;
    return f(1);
}

Original constraint:

\[ \Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor ((\neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f,\alpha,C_1[1/\alpha][\beta'/\beta]})) \]
Example

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

**Original constraint:**

\[ \Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor (\neg\alpha = 1 \land \neg\beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \]

In the previous step, we computed:

\[ \lceil \Pi_{f,\alpha,C_1} \rceil = true \]
\[ \lfloor \Pi_{f,\alpha,C_1} \rfloor = \alpha_1 \lor \lceil \Pi_{f,\alpha,C_1} \rceil [true/\alpha_1] \]
Example

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

Original constraint:

\[ \Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor (\neg\alpha = 1 \land \neg\beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])) \]

Compute greatest fixed-point:

\[
\begin{align*}
[\Pi_{f,\alpha,C_1}] &= \text{true} \\
[\Pi_{f,\alpha,C_1}] &= \alpha_1 \lor \text{false} = \alpha_1
\end{align*}
\]
Example

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

Original constraint:

\[
\Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta])
\]

Compute greatest fixed-point:

\[
[\Pi_{f,\alpha,C_1}] = true
\]
\[
[\Pi_{f,\alpha,C_1}] = \alpha_1 \lor \alpha_1[true/\alpha_1]
\]
Example

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

Original constraint:

\[
\Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor ((\neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))
\]

Compute greatest fixed-point:

\[
[\Pi_{f,\alpha,C_1}] = true \\
[\Pi_{f,\alpha,C_1}] = \alpha_1 \lor true
\]
Example

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

Original constraint:

\[
\Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor
(\neg\alpha = 1 \land \neg\beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))
\]

Compute greatest fixed-point:

\[
[\Pi_{f,\alpha,C_1}] = true
[\Pi_{f,\alpha,C_1}] = true
\]
Example

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

Original constraint:

\[
\Pi_{f,\alpha,C_1} = (\alpha = 1 \lor \beta = 2 \lor 
((\neg \alpha = 1 \land \neg \beta = 2 \land \Pi_{f,\alpha,C_1}[1/\alpha][\beta'/\beta]))
\]

Compute greatest fixed-point:

\[
[\Pi_{f,\alpha,C_1}] = true
\]
\[
[\Pi_{f,\alpha,C_1}] = true
\]

- The sufficient condition expresses that the function **MUST** return 1 because \text{VALID}([\Pi_{f,\alpha,C_1}]) holds.
The Main Result

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

- No choice variables
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 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).
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.
We also used this technique for an interprocedurally path-sensitive null dereference analysis.

<table>
<thead>
<tr>
<th></th>
<th>Interprocedurally Path-sensitive</th>
<th>Intraprocedurally Path-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenSSH 4.3p2</td>
<td>Samba 3.0.23b</td>
</tr>
<tr>
<td>Total Reports</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>Bugs</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>False Positives</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Undecided</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Report to Bug Ratio</td>
<td>3</td>
<td>2.8</td>
</tr>
</tbody>
</table>
We also used this technique for an interprocedurally path-sensitive null dereference analysis.

<table>
<thead>
<tr>
<th></th>
<th>Interprocedurally Path-sensitive</th>
<th>Intraprocedurally Path-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenSSH 4.3p2</td>
<td>Samba 3.0.23b</td>
</tr>
<tr>
<td>Total Reports</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>Bugs</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>False Positives</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Undecided</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Report to Bug Ratio</td>
<td>3</td>
<td>2.8</td>
</tr>
</tbody>
</table>
We also used this technique for an interprocedurally path-sensitive null dereference analysis.

<table>
<thead>
<tr>
<th></th>
<th>Interprocedurally Path-sensitive</th>
<th>Intraprocedurally Path-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenSSH 4.3p2</td>
<td>Samba 3.0.23b</td>
</tr>
<tr>
<td>Total Reports</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>Bugs</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>False Positives</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Undecided</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Report to Bug Ratio</td>
<td>3</td>
<td>2.8</td>
</tr>
</tbody>
</table>
We also used this technique for an interprocedurally path-sensitive null dereference analysis.

<table>
<thead>
<tr>
<th></th>
<th>Interprocedurally Path-sensitive</th>
<th>Intraprocedurally Path-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenSSH 4.3p2</td>
<td>Samba 3.0.23b</td>
</tr>
<tr>
<td>Total Reports</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>Bugs</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>False Positives</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Undecided</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Report to Bug Ratio</td>
<td>3</td>
<td>2.8</td>
</tr>
</tbody>
</table>
Experiments II

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

<table>
<thead>
<tr>
<th></th>
<th>Interprocedurally Path-sensitive</th>
<th>Intraprocedurally Path-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OpenSSH 4.3p2</td>
<td>Samba 3.0.23b</td>
</tr>
<tr>
<td>Total Reports</td>
<td>3</td>
<td>48</td>
</tr>
<tr>
<td>Bugs</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>False Positives</td>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>Undecided</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Report to Bug Ratio</td>
<td>3</td>
<td>2.8</td>
</tr>
</tbody>
</table>

- 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.

M. Das, S. Lerner, and M. Seigle.
ESP: Path-sensitive program verification in polynomial time.

Abstractions from proofs.

A. Mycroft.
Polymorphic type schemes and recursive definitions.

T. Reps, S. Horwitz, and M. Sagiv.
Precise interprocedural dataflow analysis via graph reachability.

D. Schmidt.
A calculus of logical relations for over- and underapproximating static analyses.

Y. Xie and A. Aiken.
Scalable error detection using boolean satisfiability.
Thank You For Listening!