Sound, Complete, and Scalable Path-Sensitive Analysis

Isil Dillig, Thomas Dillig, Alex Aiken
Computer Science Department
Stanford University

PLDI 2008
Motivation

- Path- and context-sensitivity add useful precision to the analysis of a large class of properties.
Motivation

- Path- and context-sensitivity add useful precision to the analysis of a large class of properties.

- Therefore, there are many proposed techniques for path- and context-sensitive program analysis.
Motivation

- Path- and context-sensitivity add useful precision to the analysis of a large class of properties.

- Therefore, there are many proposed techniques for path- and context-sensitive program analysis.
  - Model checking tools: Bebop, BLAST, SLAM, ...
Path- and context-sensitivity add useful precision to the analysis of a large class of properties.

Therefore, 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
This Talk

The Scalability Scale

Sound & Complete Scale
This Talk

Technique for path- and context-sensitive analysis that guarantees:

- soundness
- relative completeness with respect to a finite abstraction
- scales to multi-million line programs
This Talk

Technique for path- and context-sensitive analysis that guarantees:

- soundness
- relative completeness with respect to a finite abstraction
- scales to multi-million line programs

Key Insight:
This Talk

Technique for path- and context-sensitive analysis that guarantees:

- soundness
- relative completeness with respect to a finite abstraction
- scales to multi-million line programs

Key Insight:

- We can distinguish a special class of variables called unobservable variables
This Talk

Technique for path- and context-sensitive analysis that guarantees:

- soundness
- relative completeness with respect to a finite abstraction
- scales to multi-million line programs

Key Insight:

- We can distinguish a special class of variables called unobservable variables

- These variables can be eliminated from formulas used to express path-sensitive conditions without any loss of precision
  - Smaller formulas $\Rightarrow$ Better scalability
An 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);
}
```
An 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?
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);
}

\[ \Pi_{\alpha,\text{true}} = ? \]
An 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);
}
```

\[
\Pi_{\alpha,\text{true}} = (\alpha = \text{true}) \land ?
\]
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);
}

$$\Pi_{\alpha,\text{true}} = ((\alpha = \text{true}) \land (\beta = \text{'y'} \lor ?))$$
An 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);
}
```

\[ \Pi_{\alpha, \text{true}} = \exists \beta. (\alpha = \text{true} \land (\beta = 'y' \lor ?)) \]

The existential quantifier expresses:
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);
}

\[ \Pi_{\alpha, \text{true}} = \exists \beta. ((\alpha = \text{true}) \land (\beta = 'y' \lor ?)) \]

The existential quantifier expresses:

- Environment choice: We merely know that \( \beta \) has some value, i.e. it exists.
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);
}

Π_{α, true} = \exists \beta. ((α = true) \land (β = 'y' \lor ?))

The existential quantifier expresses:

- Environment choice: We merely know that \(β\) has some value, i.e. it exists.
- Scope: Each input is used for only one recursive call.
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);
}

\[\Pi_{\alpha,\text{true}} = \exists \beta. ((\alpha = \text{true}) \land (\beta = 'y' \lor ?))\]

The existential quantifier expresses:

- Environment choice: We merely know that \(\beta\) has some value, i.e. it exists.
- Scope: Each input is used for only one recursive call.
- Note: The existential has slightly non-standard semantics.
An 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);
}
```

\[
\Pi_{\alpha, true} = \exists \beta. ((\alpha = true) \land (\beta = 'y' \lor (\beta \neq 'n' \land \Pi_{\alpha, true[true/\alpha]})))
\]
An 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);
}
```

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

Π_{α, true} = \exists \beta.((α = \text{true}) \land (β = 'y' \lor (β \neq 'n' \land Π_{α, true}[\text{true}/α])))
Problem: Convergence

- If we try to solve the above constraint, we get:
Problem: Convergence

If we try to solve the above constraint, we get:

$$\Pi_{\alpha, \text{true}} = \exists \beta.(\alpha = \text{true}) \land (\beta = 'y' \lor \neg(\beta = 'n')) \land \exists \beta'.(\text{true} = \text{true}) \land (\beta' = 'y' \lor \neg(\beta' = 'n')) \land \exists \beta''.(\text{true} = \text{true}) \land (\beta'' = 'y' \lor \neg(\beta'' = 'n')) \land \ldots$$
Sound, Complete, and Scalable Path-Sensitive Analysis

Problem: Convergence

- If we try to solve the above constraint, we get:

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

- \( \exists \)-bound variables cause problems with termination.
Classification of Variables

- Observable Variables ($\alpha$)
Classification of Variables

- Observable Variables ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
Classification of Variables

- Observable Variables ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- Unobservable Variables ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input
Classification of Variables

- Observable Variables ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- Unobservable Variables ($\beta$)
Classification of Variables

- Observable Variables ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- Unobservable Variables ($\beta$)
  - $\exists$-bound variables that represent environment choices
Classification of Variables

- Observable Variables ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- Unobservable Variables ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
Classification of Variables

- **Observable Variables** ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input
Classification of Variables

- **Observable Variables** ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input
    ```java
    char userInput = getUserInput();
    if(userInput == 'y') return true;
    ```
Classification of Variables

- **Observable Variables** ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input, *system state*
Classification of Variables

- **Observable Variables** ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input, system state

```c
int* p = malloc(sizeof(int));
if(!p) return;
```
Classification of Variables

- **Observable Variables** ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input, system state, imprecision in memory abstraction
Classification of Variables

- **Observable Variables** ($\alpha$)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input, system state, imprecision in memory abstraction

```python
if(arr[i]==0) return;
```
Classification of Variables

- **Observable Variables** ($\alpha$)
  - inputs to a function provided by callers
  - e.g., arguments and globals

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input, system state, imprecision in memory abstraction

- **Return Variables** ($\Pi$)
Classification of Variables

- **Observable Variables** ($\alpha$)
  - inputs to a function provided by callers
  - e.g., arguments and globals

- **Unobservable Variables** ($\beta$)
  - $\exists$-bound variables that represent environment choices
  - Environment choice: Any variable that the user-provided abstraction cannot relate to the function inputs
  - e.g., user input, system state, imprecision in memory abstraction

- **Return Variables** ($\Pi$)
  - Represent unknowns we want to solve for
Generalized Recursive Constraints

\[
E = \begin{bmatrix}
[\Pi_{f_1,\alpha,C_i}] &=& \exists \beta_1 \cdot [\phi_{i_1}(\alpha_1, \beta_1, \Pi[b_1/\alpha])] \\
\vdots &=& \vdots \\
[\Pi_{f_k,\alpha,C_i}] &=& \exists \beta_k \cdot [\phi_{k_i}(\alpha_k, \beta_k, \Pi[b_k/\alpha])]
\end{bmatrix}
\]
Generalized Recursive Constraints

\[ E = \begin{bmatrix}
\bar{\Pi}_{f_1,\alpha,C_i} & = & \exists \beta_1 \cdot [\phi_{1i}(\bar{\alpha}_1, \beta_1, \bar{\Pi}[\bar{b}_1/\bar{\alpha}])] \\
\vdots & & \vdots \\
\bar{\Pi}_{f_k,\alpha,C_i} & = & \exists \beta_k \cdot [\phi_{ki}(\bar{\alpha}_k, \beta_k, \bar{\Pi}[\bar{b}_k/\bar{\alpha}])]
\end{bmatrix} \]
Generalized Recursive Constraints

\[ E = \begin{bmatrix}
[\tilde{\Pi}_{f_1,\alpha,C_i}] &=& \exists \tilde{\beta}_1. \ [\tilde{\phi}_{1i}(\tilde{\alpha}_1, \tilde{\beta}_1, \tilde{\Pi}[\tilde{b}_1/\tilde{\alpha}])]
\end{bmatrix}
\]

\[ \vdots \]

\[ [\tilde{\Pi}_{f_k,\alpha,C_i}] = \exists \tilde{\beta}_k. [\tilde{\phi}_{ki}(\tilde{\alpha}_k, \tilde{\beta}_k, \tilde{\Pi}[\tilde{b}_k/\tilde{\alpha}])] \]
Bad News

Unfortunately, we do not know of a way to obtain an exact solution to these constraints.
Good News

- Fortunately, for program analysis purposes, we are almost never interested in an exact solution.
Good News

- Fortunately, for program analysis purposes, we are almost never interested in an exact solution.

- Instead, as is well known, we are often interested in answering **may** and **must** queries about program properties.
Good News

- Fortunately, for program analysis purposes, we are almost never interested in an exact solution.

- Instead, as is well known, we are often interested in answering may and must queries about program properties.
  - Safety: *May* this pointer be dereferenced?
Good News

- Fortunately, for program analysis purposes, we are almost never interested in an exact solution.

- Instead, as is well known, we are often interested in answering **may** and **must** queries about program properties.
  - Safety: *May* this pointer be dereferenced?
  - Liveness: *Must* this pointer be freed?
Good News

- Fortunately, for program analysis purposes, we are almost never interested in an exact solution.

- Instead, as is well known, we are often interested in answering **may** and **must** queries about program properties.
  - Safety: *May* this pointer be dereferenced?
  - Liveness: *Must* this pointer be freed?

- To answer may queries precisely, the solution only needs to preserve **satisfiability**.
Good News

- Fortunately, for program analysis purposes, we are almost never interested in an exact solution.

- Instead, as is well known, we are often interested in answering **may** and **must** queries about program properties.
  - Safety: *May* this pointer be dereferenced?
  - Liveness: *Must* this pointer be freed?

- To answer may queries precisely, the solution only needs to preserve **satisfiability**.

- For must queries, we only need a **validity** preserving solution.
For any formula \(\phi\), the **strongest necessary condition** \(\lceil \phi \rceil\) of \(\phi\) containing *only observable variables* preserves satisfiability.

\[
\begin{align*}
(1) \quad & \phi \Rightarrow \lceil \phi \rceil \\
(2) \quad & \forall \phi'.((\phi \Rightarrow \phi') \Rightarrow (\lceil \phi \rceil \Rightarrow \phi'))
\end{align*}
\]
For any formula \( \phi \), the **strongest necessary condition** \( \lceil \phi \rceil \) of \( \phi \) containing *only observable variables* preserves satisfiability.

\[
\begin{align*}
(1) \quad \phi & \Rightarrow \lceil \phi \rceil \\
(2) \quad \forall \phi'.((\phi \Rightarrow \phi') \Rightarrow (\lceil \phi \rceil \Rightarrow \phi'))
\end{align*}
\]

Similarly, for any formula \( \phi \) the **weakest sufficient condition** \( \lfloor \phi \rfloor \) over *only observable variables* preserves validity of \( \phi \).

\[
\begin{align*}
(1) \quad \lfloor \phi \rfloor & \Rightarrow \phi \\
(2) \quad \forall \phi'.((\phi' \Rightarrow \phi) \Rightarrow (\phi' \Rightarrow \lfloor \phi \rfloor))
\end{align*}
\]
If $\phi$ is the constraint under which a program property $P$ holds, we have the following guarantees:

\[
\text{SAT}(\lceil \phi \rceil) \iff P \text{ MAY hold}
\]
\[
\text{VALID}(\lfloor \phi \rfloor) \iff P \text{ MUST hold}
\]
Example Revisited

```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);
}
```
### Example Revisited

```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);
}
```

Original constraint:

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

```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);
}
```

Original constraint:

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

Strongest Necessary Condition: $$\lceil \Pi_{\alpha,\text{true}} \rceil = (\alpha = \text{true})$$
Example Revisited

```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);
}
```

Original constraint:

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

Strongest Necessary Condition: $$[\Pi_{\alpha,\text{true}}] = (\alpha = \text{true})$$

Weakest Sufficient Condition: $$[\Pi_{\alpha,\text{true}}] = \text{false}$$
Generalized Recursive Constraints Revisited

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

Goal: Compute observable strongest necessary and weakest sufficient conditions for the solution of \( E \).
Outline of the Algorithm

- Step 0: Transform constraints to propositional formulas.
Outline of the Algorithm

- Step 0: Transform constraints to propositional formulas.
- Step 1: Eliminate the unobservable $\beta$ variables.
Outline of the Algorithm

- Step 0: Transform constraints to propositional formulas.

- Step 1: Eliminate the unobservable $\beta$ variables.

- Step 2: Transform the constraint system to preserve strongest necessary and weakest sufficient conditions under syntactic substitution.
Outline of the Algorithm

- Step 0: Transform constraints to propositional formulas.
- Step 1: Eliminate the unobservable $\beta$ variables.
- Step 2: Transform the constraint system to preserve strongest necessary and weakest sufficient conditions under syntactic substitution.
- Step 3: Solve the recursive constraints via fixed-point computation (syntactic substitution)
Step 1: Eliminate Unobservable Variables

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

\[
\text{WSC}(\phi, \beta) = \phi[\text{true}/\beta] \land \phi[\text{false}/\beta]
\]
Result of Step 1

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

\[ E_{SC} = \begin{bmatrix}
\Pi_{f_1, \alpha, C_1} &=& \phi'_{11}(\vec{\alpha}_1, [\vec{\Pi}][\vec{b}_1/\vec{\alpha}]) \\
\vdots \\
\Pi_{f_k, \alpha, C_n} &=& \phi'_{kn}(\vec{\alpha}_k, [\vec{\Pi}][\vec{b}_k/\vec{\alpha}])
\end{bmatrix} \]
Step 2: 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 2: 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 2: 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:

  - Constraints contain negated Π literals.
    But $¬[\phi] \not\Leftrightarrow [¬\phi]$ and $¬[\phi] \not\Leftrightarrow [¬\phi]$
Step 2: 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:
  
  - Constraints contain negated $\Pi$ literals.
    But $\neg[\phi] \not\iff [\neg\phi]$ and $\neg[\phi] \not\iff [\neg\phi]$

  - Implicit constraints: Existence and uniqueness
Step 2: Preservation under Syntactic Substitution I

To ensure monotonicity:

- Either replace $\neg \Pi_{f,\alpha,c}^i$ with $\bigvee_j \neg \Pi_{f,\alpha,c}^j$.
- Or use the property $\lceil \neg \phi \rceil \iff \neg \lfloor \phi \rfloor$ and $\lfloor \neg \phi \rfloor \iff \neg \lceil \phi \rceil$.

But important for a practical implementation
Step 2: Preservation under Syntactic Substitution I

- To ensure monotonicity:
  - Either replace $\neg \Pi_{f,\alpha,c_i}$ with $\bigvee_{j \neq i} \Pi_{f,\alpha,c_j}$. 

The latter requires simultaneous fixpoint computation of strongest necessary and weakest sufficient conditions, but important for a practical implementation.
Step 2: Preservation under Syntactic Substitution

To ensure monotonicity:
- 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 $\lfloor \neg \phi \rfloor \iff \neg \lceil \phi \rceil$.

But important for a practical implementation.
To ensure monotonicity:

- 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 $\lfloor \neg \phi \rfloor \iff \neg \lceil \phi \rceil$

The latter requires simultaneous fixpoint computation of strongest necessary and weakest sufficient conditions.
Step 2: Preservation under Syntactic Substitution I

- To ensure monotonicity:
  - 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 $\lfloor \neg \phi \rfloor \iff \neg \lceil \phi \rceil$

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

- But important for a practical implementation
Step 2: Preservation under Syntactic Substitution II

To eliminate implicit existence and uniqueness constraints:

- Convert to DNF and drop contradictions
  (for necessary conditions)

- Convert to CNF and drop tautologies
  (for sufficient conditions)
To eliminate implicit existence and uniqueness constraints:
  - Convert to DNF and drop contradictions
    (for necessary conditions)
  - Convert to CNF and drop tautologies
    (for sufficient conditions)

The resulting constraints preserve strongest necessary and weakest sufficient conditions under syntactic substitution.
The technique is sound and complete for answering satisfiability and validity queries with respect to some user-provided finite abstraction.
The Main Result

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

- Furthermore, since the computed strongest necessary and weakest sufficient conditions do not contain any unobservable variables, the resulting constraints are small in practice, allowing the technique to scale to large 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).
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.
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 Work

- Caveat: Previous results table excludes any error reports arising from array elements and recursive data structures.
Future Work

- Caveat: Previous results table excludes any error reports arising from array elements and recursive data structures.
  - Underlying framework collapses all unbounded data structures into one summary node
Future Work

- Caveat: Previous results table excludes any error reports arising from array elements and recursive data structures.
  
  - Underlying framework collapses all unbounded data structures into one summary node
  
  - Imprecise for verifying memory safety.
Future Work

- Caveat: Previous results table excludes any error reports arising from array elements and recursive data structures.
  - Underlying framework collapses all unbounded data structures into one summary node
  - Imprecise for verifying memory safety.
- Shape analysis is our current work-in-progress.
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.