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.

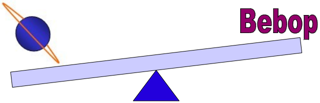
Motivation

- 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, ...

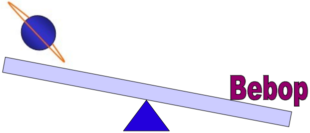
Motivation

- 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, ...
  - Lighter-weight static analysis tools: Saturn, ESP, ...

Tradeoff?

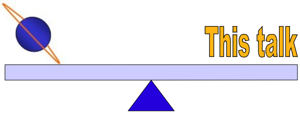


The Scalability Scale

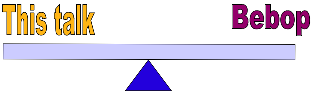


Sound & Complete Scale

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 ⇒ Better scalability

An Example

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

An Example

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

When does queryUser return true?

An Example

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

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

An Example

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

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

An Example

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

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

An Example

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

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

An Example

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

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

The existential quantifier expresses:

An Example

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

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

The existential quantifier expresses:

- Environment choice: We merely know that β has some value, i.e. it exists.

An Example

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

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

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.

An Example

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

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

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.
- Note: The existential has slightly non-standard semantics.

An Example

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

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

An Example

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

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

An Example

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

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

Problem: Convergence

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

Problem: Convergence

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

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

Problem: Convergence

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

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

- $\exists$-bound variables cause problems with termination.

Classification of Variables

- Observable Variables (α)

Classification of Variables

- Observable Variables (α)
  - caller-supplied inputs to a function, e.g., arguments and globals

Classification of Variables

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

Classification of Variables

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

Classification of Variables

- Observable Variables (α)
  - caller-supplied inputs to a function, e.g., arguments and globals
  - value is available to caller prior to invocation of this function
- Unobservable Variables (β)
  - $\exists$-bound variables that represent environment choices

Classification of Variables

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

```
char userInput = getUserInput();
if(userInput == 'y') return true;
```

Classification of Variables

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

```
int* p = malloc(sizeof(int));
if(!p) return;
```

Classification of Variables

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

- Unobservable Variables (β)
  - $\exists$-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

```
if(arr[i]==0) return;
```

Classification of Variables

- Observable Variables (α)
  - inputs to a function provided by callers
  - e.g., arguments and globals
- Unobservable Variables (β)
  - $\exists$-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 (Π)

Classification of Variables

- Observable Variables (α)
  - inputs to a function provided by callers
  - e.g., arguments and globals
- Unobservable Variables (β)
  - $\exists$-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 (Π)
  - Represent unknowns we want to solve for

Generalized Recursive Constraints

$$E = \begin{bmatrix} [\vec{\Pi}_{f_1,\alpha,C_i}] & = & \exists \vec{\beta_1}.\ [\vec{\phi}_{1i}(\vec{\alpha}_1, \vec{\beta}_1, \vec{\Pi}[\vec{b}_1/\vec{\alpha}])] \\ \vdots & & \vdots \\ [\vec{\Pi}_{f_k,\alpha,C_i}] & = & \exists \vec{\beta_k}.\ [\vec{\phi}_{ki}(\vec{\alpha}_k, \vec{\beta}_k, \vec{\Pi}[\vec{b}_k/\vec{\alpha}])] \end{bmatrix}$$

Generalized Recursive Constraints

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

Generalized Recursive Constraints

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

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.

Strongest Necessary and Weakest Sufficient Conditions

- For any formula ϕ, the **strongest necessary condition** $\lceil\phi\rceil$ of ϕ containing *only observable variables* preserves satisfiability.

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

Strongest Necessary and Weakest Sufficient Conditions

- For any formula ϕ, the **strongest necessary condition** $\lceil \phi \rceil$ of ϕ containing *only observable variables* preserves satisfiability.

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

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

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

Strongest Necessary and Weakest Sufficient Conditions (2)

If ϕ is the constraint under which a program property P holds, we have the following guarantees:

$$\text{SAT}(\lceil \phi \rceil) \Leftrightarrow \text{P } \textbf{MAY} \text{ hold}$$

$$\text{VALID}(\lfloor \phi \rfloor) \Leftrightarrow \text{P } \textbf{MUST} \text{ hold}$$

Example Revisited

```
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

```
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:

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

Example Revisited

```
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:

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

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

Example Revisited

```
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:

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

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

Weakest Sufficient Condition: $\lfloor \Pi_{\alpha,\text{true}} \rfloor = \text{false}$

Generalized Recursive Constraints Revisited

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

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 β variables.

Outline of the Algorithm

- Step 0: Transform constraints to propositional formulas.
- Step 1: Eliminate the unobservable β variables.
- Step 2: Transform the constraint system to preserve strongest necessary and weakest sufficient conditions under syntactic substitution.

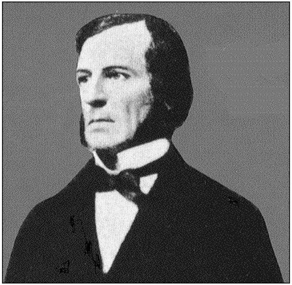
Outline of the Algorithm

- Step 0: Transform constraints to propositional formulas.
- Step 1: Eliminate the unobservable β 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

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

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



Result of Step 1

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

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

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 $\neg\lceil\phi\rceil \not\Leftrightarrow \lceil\neg\phi\rceil$ and $\neg\lfloor\phi\rfloor \not\Leftrightarrow \lfloor\neg\phi\rfloor$

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 $\neg\lceil\phi\rceil \not\Leftrightarrow \lceil\neg\phi\rceil$ and $\neg\lfloor\phi\rfloor \not\Leftrightarrow \lfloor\neg\phi\rfloor$

  - Implicit constraints: Existence and uniqueness

Step 2: Preservation under Syntactic Substitution I

- To ensure monotonicity:

Step 2: Preservation under Syntactic Substitution I

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

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 $\lceil\neg\phi\rceil \Leftrightarrow \neg\lfloor\phi\rfloor$ and $\lfloor\neg\phi\rfloor \Leftrightarrow \neg\lceil\phi\rceil$

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 $\lceil\neg\phi\rceil \Leftrightarrow \neg\lfloor\phi\rfloor$ and $\lfloor\neg\phi\rfloor \Leftrightarrow \neg\lceil\phi\rceil$
- The latter requires simultaneous fixpoint computation of strongest necessary and weakest sufficient conditions

Step 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 $\lceil\neg\phi\rceil \Leftrightarrow \neg\lfloor\phi\rfloor$ and $\lfloor\neg\phi\rfloor \Leftrightarrow \neg\lceil\phi\rceil$
- The latter requires simultaneous fixpoint computation of strongest necessary and weakest sufficient conditions
- But important for a practical implementation

Step 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)

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

The Main Result

Main Result

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

The Main Result

Main Result

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

- Furthermore, since the computed strongest necesssary 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).

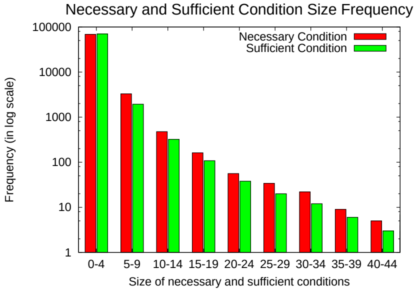
Experiments I

- 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.

Experiments II

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

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

Experiments II

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

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

Experiments II

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

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

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

Future 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.
In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130, London, UK, 2000. Springer-Verlag.

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

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

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

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

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

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