Automated Error Diagnosis Using Abductive Inference

Isil Dillig[1] Thomas Dillig[1] Alex Aiken[2]

[1]Department of Computer Science
College of William & Mary, Virginia, USA

[2]Department of Computer Science
Stanford University, CA, USA

PLDI 2012

Severin Heiniger

An Ordinary Day in a Developer's Life

```c
void foo(int flag, unsigned int n) {
  int k = 0, i = 0, j = 0, z = 0;
  if (flag) k = n;
  else      k = 1;

  while (i <= n) {
    i = i + 1;
    j = j + i;
  }
  int z = k + i + j;
  assert(z > 2 * n);
}
```

An Ordinary Day in a Developer's Life

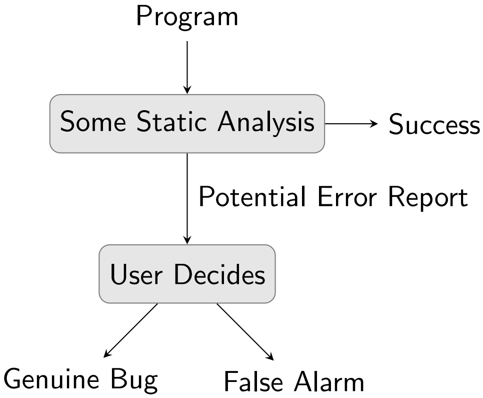
```
void foo(int flag, unsigned int n) {
  int k = 0, i = 0, j = 0, z = 0;
  if (flag) k = n;
  else      k = 1;

  while (i <= n) {
    i = i + 1;
    j = j + i;
  }
  int z = k + i + j;
  assert(z > 2 * n);
}
```

Static analysis tool error report

Assertion $z > 2 * n$ may not always hold.

Manual Report Classification



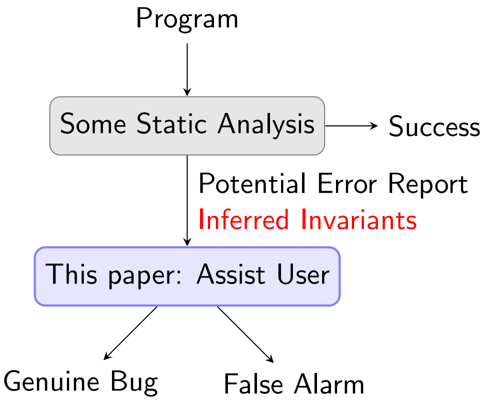
Manual Report Classification

- Time-consuming
- User repeats all successful reasoning by tool
- Error-prone

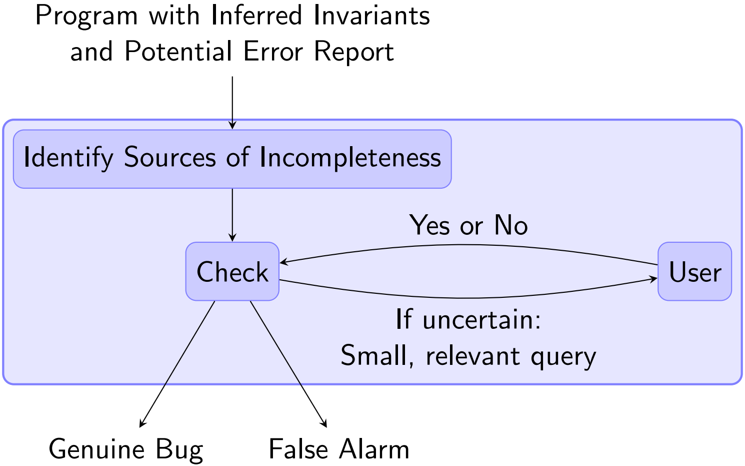
Effect

Major impediment to adoption of static analysis tools

Semi-Automated Report Classification



Semi-Automated Report Classification



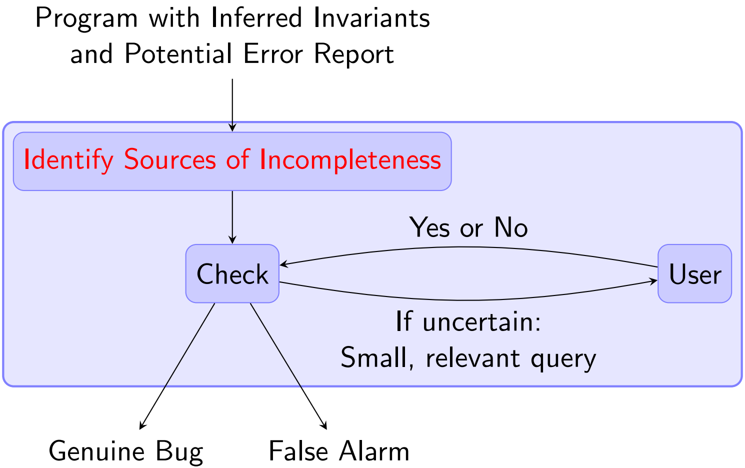
Queries

- Proof Obligation Query: *Is property P an invariant?*
  - If yes, the program is certainly error-free (false alarm)
- Failure Witness Query: *Can property P arise in some execution?*
  - If yes, the program is certainly buggy

Strategy

Pose queries in order of increasing cost (easiest first) to minimize the amount of trusted information the user must supply

We are Here


Program with Inferred Invariants
and Potential Error Report
Identify Sources of Incompleteness
Check
Yes or No
User
If uncertain:
Small, relevant query
Genuine Bug
False Alarm

Input

- Program with parameters, local variables, conditionals and while loops
- Only linear arithmetic, no function calls
- While loops annotated with inferred post-condition p':
  while(p) { s } [p']
- Program ends with an assert(p)

Identify Sources of Incompleteness

Symbolically evaluate the program. At each point in the program, environment $\mathbb{S}$ maps program variables to *symbolic value sets*.

$\mathbb{S}(i) = \{\ldots, (\pi, \phi), \ldots\}$	Under constraint ϕ, the value of variable i is the symbolic expression π

Constraints ϕ keep values from different paths separate. π can contain

Input Variables ν	For unknown program inputs
Abstraction Variables α	For unknown values due to imprecisions, e.g., after loops

Example

```
void foo(int flag, unsigned int n) {
  int k = 0, i = 0, j = 0, z = 0;
```

$$\mathbb{S}(k) = \{(0, true)\} \quad \mathbb{S}(i) = \{(0, true)\} \quad \ldots$$

```
  if (flag) k = n;
  else      k = 1;
```

$$\mathbb{S}(k) = \{(1, \neg\nu_{flag}), (\nu_n, \nu_{flag})\}$$

```
  while (i <= n) {
    i = i + 1;
    j = j + i;
  }
```

$$\mathbb{S}(i) = \{(\alpha_i, true)\} \quad \mathbb{S}(j) = \{(\alpha_j, true)\}$$

```
  int z = k + i + j;
```

$$\mathbb{S}(z) = \{(1 + \alpha_i + \alpha_j, \neg\nu_{flag}), (\nu_n + \alpha_i + \alpha_j, \nu_{flag})\}$$

```
  assert(z > 2*n);
}
```

Example

```
void foo(int flag, unsigned int n) {
  int k = 0, i = 0, j = 0, z = 0;
                                          𝕊(k) = {(0, true)}   𝕊(i) = {(0, true)}   ...
  if (flag) k = n;
  else      k = 1;
                                          𝕊(k) = {(1, ¬ν_flag), (ν_n, ν_flag)}
  while (i <= n) {
    i = i + 1;
    j = j + i;
  } [i ≥ 0 ∧ i > n]                       𝕊(i) = {(α_i, true)}   𝕊(j) = {(α_j, true)}
  int z = k + i + j;                      𝕊(z) = {(1 + α_i + α_j, ¬ν_flag), (ν_n + α_i + α_j, ν_flag)}
  assert(z > 2 * n);
}
```

Propagate inferred invariants as constraints on abstract variables

$$\mathcal{I} = (\alpha_i \geq 0 \wedge \alpha_i > \nu_n \wedge \nu_n \geq 0)$$

Example

```
void foo(int flag, unsigned int n) {
  int k = 0, i = 0, j = 0, z = 0;
```

$$\mathbb{S}(k) = \{(0, \mathit{true})\} \quad \mathbb{S}(i) = \{(0, \mathit{true})\} \quad \ldots$$

```
  if (flag) k = n;
  else      k = 1;
```

$$\mathbb{S}(k) = \{(1, \neg\nu_{\mathit{flag}}), (\nu_n, \nu_{\mathit{flag}})\}$$

```
  while (i <= n) {
    i = i + 1;
    j = j + i;
  } [i ≥ 0 ∧ i > n]
  int z = k + i + j;
  assert(z > 2 * n);
}
```

$$\mathbb{S}(i) = \{(\alpha_i, \mathit{true})\} \quad \mathbb{S}(j) = \{(\alpha_j, \mathit{true})\}$$

$$\mathbb{S}(z) = \{(1 + \alpha_i + \alpha_j, \neg\nu_{\mathit{flag}}), (\nu_n + \alpha_i + \alpha_j, \nu_{\mathit{flag}})\}$$

Symbolically evaluate the assertion predicate

$$\phi = (1 + \alpha_i + \alpha_j > 2 * \nu_n \wedge \neg\nu_{\mathit{flag}}) \vee (\nu_n + \alpha_i + \alpha_j > 2 * \nu_n \wedge \nu_{\mathit{flag}})$$

Result

The result is a pair of symbolic constraints

- $\mathcal{I}$ All known invariants on abstract variables
- ϕ Condition under which the assertion evaluates to *true*

Result

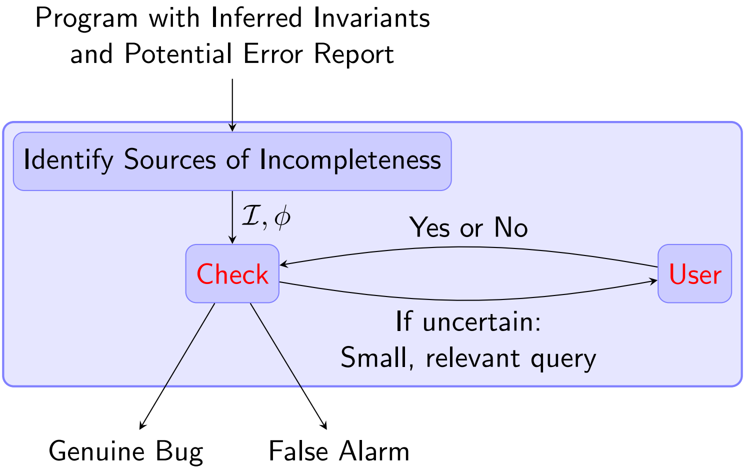
The result is a pair of symbolic constraints

- $\mathcal{I}$ All known invariants on abstract variables
- ϕ Condition under which the assertion evaluates to *true*

Lemma

If $\mathcal{I} \models \phi$, *then the program is error-free (assertion always succeeds)*
If $\mathcal{I} \models \neg\phi$, *then the program must be buggy (assertion always fails)*

We are Here


Program with Inferred Invariants
and Potential Error Report
Identify Sources of Incompleteness
$\mathcal{I}, \phi$
Yes or No
Check
User
If uncertain:
Small, relevant query
Genuine Bug
False Alarm

Proof Obligation

Given known facts $\mathcal{I}$ and success condition ϕ, a *proof obligation* is a formula Γ that – together with $\mathcal{I}$ – proves ϕ:

$$\Gamma \wedge \mathcal{I} \models \phi \quad \text{and} \quad SAT(\Gamma \wedge \mathcal{I})$$

Proof Obligation

Given known facts $\mathcal{I}$ and success condition ϕ, a *proof obligation* is a formula Γ that – together with $\mathcal{I}$ – proves ϕ:

$$\Gamma \wedge \mathcal{I} \models \phi \quad \text{and} \quad \mathit{SAT}(\Gamma \wedge \mathcal{I})$$

$\mathit{Cost}(\Gamma)$

$$\begin{aligned} & 1 \cdot \#\,\text{abstraction variables } \alpha \in \mathit{Vars}(\Gamma) \\ + & |\mathit{Vars}(\phi) \cup \mathit{Vars}(\mathcal{I})| \cdot \#\,\text{input variables } \nu \in \mathit{Vars}(\Gamma) \end{aligned}$$

- The fewer variables, the better
- No input variables if possible

Failure Witness

Given known facts $\mathcal{I}$ and success condition ϕ, a *failure witness* is a formula Υ that – together with $\mathcal{I}$ – proves $\neg\phi$:

$$\Upsilon \wedge \mathcal{I} \models \neg\phi \quad \text{and} \quad \mathit{SAT}(\Upsilon \wedge \mathcal{I})$$

$\mathit{Cost}(\Upsilon)$

$$\begin{aligned}|\mathit{Vars}(\phi) \cup \mathit{Vars}(\mathcal{I})| \cdot \#\,&\text{abstraction variables } \alpha \in \mathit{Vars}(\Upsilon)\\ + 1 \cdot \#\,&\text{input variables } \nu \in \mathit{Vars}(\Upsilon)\end{aligned}$$

- The fewer variables, the better
- Prefer input variables

Weakest Minimum Queries

Weakest Minimum Proof Obligation Γ

- costs less than or equal to any other proof obligation, and
- is no stronger than any other proof obligations with same cost

Weakest Minimum Failure Witness Υ Dito

Ask the User

Ask the user the one with lower cost

- *Does* Γ *hold in all program executions?*
  - Yes Program is error-free (because $\Gamma \wedge \mathcal{I} \models \phi$)
  - No Add $\neg\Gamma$ to known witnesses and maybe ask another query
- *May* Υ *arise in some execution?*
  - Yes Programm is buggy (because $\Upsilon \wedge \mathcal{I} \models \neg\phi$)
  - No Add $\neg\Upsilon$ to known facts $\mathcal{I}$ and maybe ask another query

Example

```c
void foo(int flag, unsigned int n) {
  int k = 0, i = 0, j = 0, z = 0;
  if (flag) k = n;
  else      k = 1;

  while (i <= n) {
    i = i + 1;
    j = j + i;
  }
  int z = k + i + j;
  assert(z > 2 * n);
}
```

$$\mathcal{I} = (\alpha_i \geq 0 \wedge \alpha_i > \nu_n \wedge \nu_n \geq 0)$$

$$\phi = (1 + \alpha_i + \alpha_j > 2 * \nu_n \wedge \neg \nu_{flag}) \vee (\nu_n + \alpha_i + \alpha_j > 2 * \nu_n \wedge \nu_{flag})$$

Weakest Minimum Proof Obligation $\Gamma = (\alpha_j \geq \nu_n)$

Weakest Minimum Failure Witness $\Upsilon = (\neg \nu_{flag} \wedge \alpha_i + \alpha_j < 0)$

Example

```c
void foo(int flag, unsigned int n) {
  int k = 0, i = 0, j = 0, z = 0;
  if (flag) k = n;
  else      k = 1;

  while (i <= n) {
    i = i + 1;
    j = j + i;
  }
  int z = k + i + j;
  assert(z > 2 * n);
}
```

$$\mathcal{I} = (\alpha_i \geq 0 \wedge \alpha_i > \nu_n \wedge \nu_n \geq 0)$$

$$\phi = (1 + \alpha_i + \alpha_j > 2 * \nu_n \wedge \neg\nu_{flag}) \vee (\nu_n + \alpha_i + \alpha_j > 2 * \nu_n \wedge \nu_{flag})$$

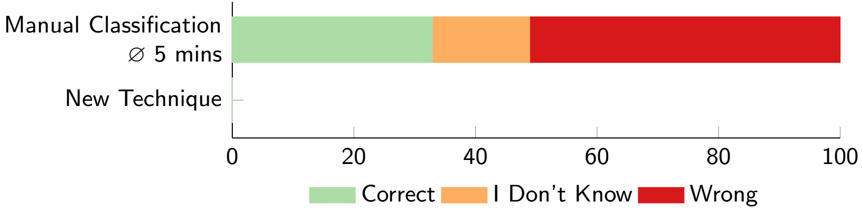
Weakest Minimum Proof Obligation $\Gamma = (\alpha_j \geq \nu_n)$ ✔ (false alarm!)

Weakest Minimum Failure Witness $\Upsilon = (\neg\nu_{flag} \wedge \alpha_i + \alpha_j < 0)$

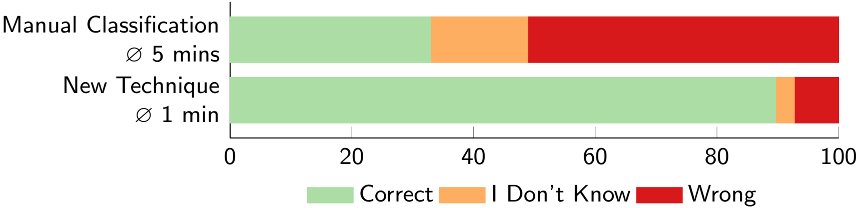
User Study: Setup

- 56 professional C programmers
- Classify 11 uncertain error reports for real-world code as
  - Genuine bugs (5), or
  - False alarms (6), or
  - *I don't know*
- Randomly assigned to classify manually or using the new technique

User Study: Results



User Study: Results



Related Work

Explaining Error Traces in Model Checking

Requires counter-example, does not address false alarms

Counterexample-Guided Abstraction Refinement (CEGAR)

Learn new predicates from concrete counter-example trace

Fully automatic, but not guaranteed to terminate

Conclusion

- Implementation not (yet) publicly available
- Practical technique to help programmers classify error reports
- Tool-agnostic

Questions

Language

$$
\begin{array}{lcl}
\text{Program } P & := & \lambda\vec{a}.\,(\texttt{let } \vec{v} \texttt{ in } (s; \texttt{check}(p))) \\
\text{Statement } s & := & v = e \mid \texttt{skip} \mid s_1; s_2 \\
 & & \mid \texttt{if}(p) \texttt{ then } s_1 \texttt{ else } s_2 \\
 & & \mid \texttt{while}^{\rho}(p)\{s\}[@p']? \\
\text{Expression } e & := & v \mid c \mid c * e \mid e_1 \oplus e_2 \;\; (\oplus \in \{+, -\}) \\
\text{Predicate } p & := & e_1 \oslash e_2 \;\; (\oslash \in \{<, >, =\}) \\
 & & \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid \neg p
\end{array}
$$

Operational Semantics of the Language

$$\frac{}{S \vdash v : S(v)} \qquad \frac{}{S \vdash c : c} \qquad \frac{\oplus \in \{+, -, *\} \quad S \vdash e_1 : c_1 \;\; S \vdash e_2 : c_2}{S \vdash e_1 \oplus e_2 : c_1 \oplus c_2}$$

$$\frac{S \vdash e_1 : c_1 \;\; S \vdash e_2 : c_2 \quad b = \begin{cases} \text{true} & \text{if } c_1 \oslash c_2 \\ \text{false} & \text{otherwise} \end{cases}}{S \vdash e_1 \oslash e_2 : b} \qquad \frac{\text{lop} \in \{\wedge, \vee\} \quad S \vdash p_1 : b_1 \;\; S \vdash p_2 : b_2}{S \vdash p_1 \;\text{lop}\; p_2 : b_1 \;\text{lop}\; b_2}$$

$$\frac{S \vdash p : b}{S \vdash \neg p : \neg b} \qquad \frac{S \vdash e : c}{S \vdash v = e : S[c/v]} \qquad \frac{}{S \vdash \mathtt{skip} : S}$$

$$\frac{S \vdash p : \text{true} \quad S \vdash s_1 : S_1}{S \vdash \mathtt{if}(p)\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 : S_1} \qquad \frac{S \vdash p : \text{false} \quad S \vdash s_2 : S_2}{S \vdash \mathtt{if}(p)\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2 : S_2}$$

$$\frac{S \vdash s_1 : S_1 \quad S_1 \vdash s_2 : S_2}{S \vdash s_1; s_2 : S_2} \qquad \frac{S \vdash p : \text{true} \quad S \vdash s : S' \quad S' \vdash \text{loop}^{\rho}(p)\{s\} : S''}{S \vdash \text{loop}^{\rho}(p)\{s\} : S''}$$

$$\frac{S \vdash \text{loop}^{\rho}(p)\{s\} : S' \quad S' \vdash p' : \text{true}}{S \vdash \mathtt{while}^{\rho}(p)\{s\}[@p'] : S'} \qquad \frac{S \vdash p : \text{false}}{S \vdash \text{loop}^{\rho}(p)\{s\} : S}$$

$$\frac{S = [c_1/a_1, \ldots, c_k/a_k][0/v_1, \ldots, 0/v_n] \quad S \vdash s : S' \quad S' \vdash p : b}{\vdash \lambda\vec{a}.(\mathtt{let}\ \vec{v}\ \mathtt{in}\ (s; \mathtt{check}(p)))(c_1, \ldots c_k) : b}$$

Operations on Symbolic Value Sets

$$\frac{\begin{array}{l}\theta_1 = \{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\} \\ \theta_2 = \{(\pi'_1, \phi'_1), \ldots, (\pi'_n, \phi'_n)\} \\ \theta = \bigcup_{ij} ((\pi_i \oplus \pi'_j), (\phi_i \wedge \phi'_j))\end{array}}{\vdash \theta_1 \oplus \theta_2 : \theta}$$

$$\frac{\begin{array}{l}\theta_1 = \{(\pi_1, \phi_1), \ldots, (\pi_k, \phi_k)\} \\ \theta_2 = \{(\pi'_1, \phi'_1), \ldots, (\pi'_n, \phi'_n)\} \\ \phi = \bigvee_{ij} ((\pi_i \oslash \pi'_j) \wedge \phi_i \wedge \phi'_j)\end{array}}{\vdash \theta_1 \oslash \theta_2 : \phi}$$

$$\frac{\theta' = \bigcup_{(\pi_i, \phi_i) \in \theta} (\pi_i, (\phi_i \wedge \phi))}{\vdash \theta \wedge \phi : \theta'}$$

Symbolic Evaluation Rules for Expressions and Predicates

$$\frac{}{\mathbb{S} \vdash v : \mathbb{S}(v)} \qquad \frac{}{\mathbb{S} \vdash c : (c, \mathit{true})} \qquad \frac{\oplus \in \{+, -, *\} \quad \mathbb{S} \vdash e_1 : \theta_1 \quad \mathbb{S} \vdash e_2 : \theta_2}{\mathbb{S} \vdash e_1 \oplus e_2 : \theta_1 \oplus \theta_2}$$

$$\frac{\mathbb{S} \vdash e_1 : \theta_1 \quad \mathbb{S} \vdash e_2 : \theta_2}{\mathbb{S} \vdash e_1 \oslash e_2 : \theta_1 \oslash \theta_2} \qquad \frac{\text{lop} \in \{\wedge, \vee\} \quad \mathbb{S} \vdash p_1 : \phi_1 \quad \mathbb{S} \vdash p_2 : \phi_2}{\mathbb{S} \vdash p_1 \ \text{lop} \ p_2 : \phi_1 \ \text{lop} \ \phi_2} \qquad \frac{\mathbb{S} \vdash p : \phi}{\mathbb{S} \vdash \neg p : \neg \phi}$$

Transformers for the Symbolic Evaluation

$$\frac{\mathbb{S} \vdash e : \theta \quad \mathbb{S}' = \mathbb{S}[\theta/v]}{\mathbb{S}, \mathcal{I} \vdash v = e : \mathbb{S}', \mathcal{I}} \qquad \frac{}{\mathbb{S}, \mathcal{I} \vdash \texttt{skip} : \mathbb{S}, \mathcal{I}} \qquad \frac{\mathbb{S}, \mathcal{I} \vdash s_1 : \mathbb{S}_1, \mathcal{I}_1 \quad \mathbb{S}_1, \mathcal{I}_1 \vdash s_2 : \mathbb{S}_2, \mathcal{I}_2}{\mathbb{S}, \mathcal{I} \vdash s_1; s_2 : \mathbb{S}_2, \mathcal{I}_2}$$

$$\frac{\begin{array}{c} \mathbb{S} \vdash p : \phi \\ \mathbb{S}, \mathcal{I} \vdash s_1 : \mathbb{S}_1, \mathcal{I}_1 \qquad \mathbb{S}, \mathcal{I} \vdash s_2 : \mathbb{S}_2, \mathcal{I}_2 \\ \mathbb{S}' = (\mathbb{S}_1 \wedge \phi) \sqcup (\mathbb{S}_2 \wedge \neg\phi) \\ \mathcal{I}' = ((\phi \Rightarrow \mathcal{I}_1) \wedge (\neg\phi \Rightarrow \mathcal{I}_2)) \end{array}}{\mathbb{S}, \mathcal{I} \vdash \texttt{if}(p) \texttt{ then } s_1 \texttt{ else } s_2 : \mathbb{S}', \mathcal{I}'}$$

$$\frac{\mathbb{S}' = \mathbb{S}[(\alpha_1^\rho, \mathit{true})/v_1, \ldots, (\alpha_k^\rho, \mathit{true})/v_k])(\vec{v} \text{ modified in } s)}{\mathbb{S}, \mathcal{I} \vdash \text{loop}^\rho(p)\{s\} : \mathbb{S}', \mathcal{I}}$$

$$\frac{\mathbb{S}, \mathcal{I} \vdash \text{loop}^\rho(p)\{s\} : \mathbb{S}', \mathcal{I} \qquad \mathbb{S}' \vdash p' : \phi}{\mathbb{S}, \mathcal{I} \vdash \texttt{while}^\rho(p)\{s\}[@p'] : \mathbb{S}', \mathcal{I} \wedge \phi}$$

$$\frac{\begin{array}{l} \mathbb{S} = [(\nu_1, \mathit{true})/a_1, \ldots, (\nu_k, \mathit{true})/a_k] \\ \mathbb{S}' = \mathbb{S}[(0, \mathit{true})/v_1, \ldots, (0, \mathit{true})/v_n] \\ \mathbb{S}', \mathit{true} \vdash s : \mathbb{S}'', \mathcal{I} \qquad \mathbb{S}'' \vdash p : \phi \end{array}}{\vdash \lambda\vec{a}.(\texttt{let } \vec{v} \texttt{ in } (s; \texttt{check}(p))) : \mathcal{I}, \phi}$$

Definitions for Proof Obligations I

Proof Obligation

Given known facts $\mathcal{I}$ and success condition ϕ, a *proof obligation* is a formula Γ such that

$$\Gamma \wedge \mathcal{I} \models \phi \quad \text{and} \quad SAT(\Gamma \wedge \mathcal{I})$$

Cost of Proof Obligation

Let Γ be a proof obligation query for $\mathcal{I}, \phi$, and let Π_p be a mapping from variables to costs such that $\Pi_p(\alpha) = 1$ for abstraction variable α and $\Pi_p(\nu) = |Vars(\phi) \cup Vars(\mathcal{I})|$ for input variable ν. Then,

$$Cost(\Gamma) = \sum_{v \in Vars(\Gamma)} \Pi_p(v)$$

Definitions for Proof Obligations II

Weakest Minimum Proof Obligation

Given known facts $\mathcal{I}$ and success condition ϕ, a *weakest minimum proof obligation* is a formula Γ such that

1. $\Gamma \wedge \mathcal{I} \models \phi \quad \text{and} \quad SAT(\Gamma \wedge \mathcal{I})$
2. For any other Γ' that satisfies 1, either $Cost(\Gamma) < Cost(\Gamma')$ or $Cost(\Gamma) = Cost(\Gamma') \wedge (\Gamma \not\Rightarrow \Gamma' \vee \Gamma \Leftrightarrow \Gamma')$

Computing Weakest Minimum Proof Obligations

First, rewrite $\Gamma \wedge \mathcal{I} \models \phi$ as $\Gamma \models \mathcal{I} \Rightarrow \phi$.

Cost of Partial Assignment

Let σ be a partial assignment for a formula ϕ and let Π be a mapping from variables in ϕ to non-negative integers. The cost of partial assignment σ is

$$Cost(\sigma) = \sum_{v \in Vars(\sigma)} \Pi(v)$$

Minimum Satisfying Assignment

Given mapping Π from variables to costs, a minimum satisfying assignment of formula φ is a partial assignment σ to a subset of the variables in φ such that

- $\sigma(\varphi) \equiv true$
- $\forall \sigma'$ such that $\sigma'(\varphi) \equiv true$, $Cost(\sigma) \leq Cost(\sigma')$

Computing Weakest Minimum Proof Obligations II

Minimum statisfying assignments help determine the minimum set of variables that any proof obligation Γ must contain.

Consistent Minimum Satisfying Assignment

A minimum satisfying assignment σ of φ is consistent with φ' if $\sigma(\varphi')$ is satisfiable.

Assignments that falsify $\mathcal{I}$ are not interesting. We want a minimum statisfying assignment to $\mathcal{I} \Rightarrow \phi$ that is consistent with $\mathcal{I}$.

Interpret σ as a logical formula F_σ. F_σ is a *strongest* proof obligation. It assigns each variable to a concrete value.

Computing Weakest Minimum Proof Obligations III

We want the *weakest sufficient condition* of $\mathcal{I} \Rightarrow \phi$ containing only variables in σ.

Lemma

Let V be the set of variables in a minimum satisfying assignment of $\mathcal{I} \Rightarrow \phi$ consistent with $\mathcal{I}$, and let $\overline{V}$ be the set of variables in $\mathcal{I} \Rightarrow \phi$ but not in V. We can obtain a weakest minimum proof obligation by eliminating the quantifiers from the formula

$$\forall \overline{V}. (\mathcal{I} \Rightarrow \phi)$$

Deciding Proof Obligation Queries

Valid Answer to Proof Obligation Query

We say that the answer to a proof obligation query Γ is valid iff:

- The answer is either yes or no
- If the answer is yes, then Γ holds on *all* program executions (i.e., Γ is a program invariant)
- If the answer is no, then there is at least one execution in which Γ is violated

Lemma

Let Γ be a proof obligation query and suppse yes is a valid answer to this query. Then, the program is error-free.

From Formulas to Queries

- Translate analysis variables into program expressions (easy)
- Decompose complex queries to a series of simpler queries
  - If $\phi_1 \wedge \phi_2$ is an invariant, so are ϕ_1 and ϕ_2
  - If $\phi_1 \vee \phi_2$ is a witness, so are ϕ_1 and ϕ_2
  - Convert invariant queries to CNF and witness queries to DNF
  - Treat each clause as separate, independent query
- We learn additional facts for every subquery

Algorithm (Given $\mathcal{I}$ and ϕ)

$W := \emptyset$
while (true) {
    if (Valid($\mathcal{I} \Rightarrow \phi$)) **return** ERROR_DISCHARGED
    if ($\exists \psi \in W.\,\text{UNSAT}(\mathcal{I} \wedge \psi \wedge \phi)$) **return** ERROR_VALIDATED
    $V_1 = \text{ComputeMSA}(\mathcal{I} \Rightarrow \phi, W \cup \mathcal{I}, \Pi_p)$
    $\Gamma = \text{ElimQuantifier}(\forall \overline{V_1}.\,(\mathcal{I} \Rightarrow \phi))$
    $V_2 = \text{ComputeMSA}(\mathcal{I} \Rightarrow \neg\phi, W \cup \mathcal{I}, \Pi_w)$
    $\Upsilon = \text{ElimQuantifier}(\forall \overline{V_2}.\,(\mathcal{I} \Rightarrow \neg\phi))$

    if ($\text{Cost}(\Gamma) < \text{Cost}(\Upsilon)$) {
        $Q_1 = \text{FormInvariantQuery}(\Gamma)$
        if (answer to Q_1 = YES) **return** ERROR_DISCHARGED
        $W := W \cup \neg\Gamma$
    } **else** {
        $Q_2 = \text{FormWitnessQuery}(\Upsilon)$
        if (answer to Q_2 = YES) **return** ERROR_VALIDATED
        $\mathcal{I} := \mathcal{I} \wedge \neg\Upsilon$
    }
}

Implementation

- Implemented on top of Compass analysis framework for C programs
- Also reasons about heap objects, arrays and function calls
- Sources of imprecisions are loops, non-linear arithmetic, inline assembly, etc.
- Allow the user to answer *I don't know*
- Uses own Mistral SMT solver to compute minimum satisfying assignments.

References

Isill Dillig, Thomas Dillig and Alex Aiken.
Automated Error Diagnosis Using Abductive Inference
Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), 181–192, 2012.