From Uncertainty to Belief: Inferring the Specification Within

Stephen McLaughlin
Overview

- **Area**: Program analysis and error checking / program specification
- **Problem**:
  - Tools lack adequate specification.
  - Good specifications are hard to make.
  - More specifically, the ownership problem
- **Solution**: Automate some or all of program specification
- **Methodology**: Quasi expert systems approach / implementation
Uses of program analysis

1. Basic syntax, convention and common error checking - lint
2. Slightly more advanced error detection, e.g. dereferencing NULL and type checking - splint
3. Formal methods

What is the trend moving down this list?
Annotations

- **splint**
  - `/*@null*/ char *c` - Forces a check for NULL before every dereference
  - `/*@in*/ int *i` - Forces an actual parameter to be completely defined before being passed to a function
  - `/*@out*/ int *o` - Forces the parameter to be completely defined before the function returns

- Java 5.0+ Supports basic pre-defined annotations, as well as some advanced features
  - Users can define their own annotations, which can themselves be annotated.
  - A method’s annotations can be obtained at runtime through Java reflection.
  - `@Override someMethod()` - Compiler throws an error if the annotated method does not override one in a superclass
Basic idea: Who is allocating & deallocating memory

Generalized: Who is returning and claiming ownership of resources?

Ownership: A pointer owns a resource if it is the pointer that could currently be used to de-allocate that resource.

Possible annotations for a function are
  - \( co \) - Claims ownership of a resource
  - \( ro \) - Returns ownership
  - \( \neg ro / \neg co \) - ?

Assumption?
Syntax: `method:ret|formal parameter number`
Example:

```c
FILE *fp = fopen("myfile.txt","r");
fread(buffer, n, 1000, fp);
fclose(fp);
```
fopen: $\text{ret} \in \{\neg ro, ro\}$

```c
FILE *fp = fopen("myfile.txt","r");
```

fread: $4 \in \{\neg co, co\}$

```c
fread(buffer, n, 1000, fp);
```

fclose: $1 \in \{\neg co, co\}$

```c
fclose(fp);
```
Specifications and Factors

- $A$ is the set of annotation variables in a program
- $A = a$ is an assignment to these variables, that is, a specification
- A factor $f_i$ is a mapping:
  - $f_i : A_i \rightarrow [0, \infty)$
  - where $A_i \subseteq A$.
- Example:
  - $f_{FILE}(fopen : ret = ro) = 0.51$ and $f_{FILE}(fopen : ret = \neg ro) = 0.49$
- Product of experts
  - $P(A) = \frac{1}{Z} \prod_{f_i \in \{f_i\}} f_i(A)$
More factors

- **Prior Factors:**
  - One per annotation variable
  - Adds a bias to each variable

- **Check Factors:**
  - Two values that sum to 1
  - $\theta_{OK}$ and $\theta_{BUG}$
  - OK or BUG is decided by a FSM
  - Normally want $\theta_{OK} > \theta_{BUG}$

- Where does each type of factor come from?
Annotation Factor Graph (AFG)

The code:

1. FILE * fp1 = fopen("myfile.txt", "r");
2. FILE * fp2 = fdopen(fd, "w");
3. fread(buffer, n, 1, fp1);
4. fwrite(buffer, n, 1, fp2);
5. fclose(fp1);
6. fclose(fp2);

The graph:
Using the graph

The graph illustrates the relationships between various functions, with arrows indicating dependencies or interactions:

- `f_{ro}`
- `f_{co}`
- `f_{check}`

Key nodes are:
- `fopen: ret`
- `fread: 4`
- `fclose: 1`
- `fdopen: ret`
- `fwrite: 4`
More inference techniques

**Multiple Behavioral Tests**
- Adds new states to FSM used for check factors
- Correct States: Deallocator, Ownership, Contra-Ownership
- Incorrect States: Leak, Invalid Use
- Once again, correct states weighted more heavily than incorrect

**Naming conventions**
- Most developers follow a pattern in which functions with similar behaviors or purposes have similar names.
- We add an extra factor to each callsite that evaluates to $\theta\langle\text{keyword},(co|ro)\rangle$ or $\theta\langle\text{keyword},(\neg co|\neg ro)\rangle$
- Note that developers can influence the accuracy of this factor by choice of keywords.
Implementation

- Checker observes
  - All call sites that return pointer
  - String constants (treated as returned by \( \neg \text{ro} \)) (Why?)
  - Pointer dereferences (treated as \( \neg \text{co} \)) (Why?)

- Note: Model is computationally infeasible, so Gibb’s sampling is used.

- Also uses, simulated annealing and false path pruning for AFGs
## Evaluation

| Codebase | Lines ($10^3$) | $|A|$ | # Checks | $ro$ | $\neg ro$ | $\frac{r_o}{\neg r_o}$ | $co$ | $\neg co$ | $\frac{c_o}{\neg c_o}$ | Total |
|----------|----------------|------|----------|------|----------|------------------------|------|----------|------------------------|-------|
| SDL      | 51.5           | 843  | 577      | 35   | 25       | 1.4               | 16   | 31       | 0.51                   | 107   |
| OpenSSH  | 80.12          | 717  | 3416     | 45   | 28       | 1.6               | 10   | 108      | 0.09                   | 191   |
| GIMP     | 568.3          | 4287 | 21478    | 62   | 24       | 2.58              | 7    | 109      | 0.06                   | 202   |
| XNU      | 1381.1         | 1936 | 9169     | 35   | 49       | 0.71              | 17   | 99       | 0.17                   | 200   |
| Linux    | 6580.3         | 10736| 92781    | 21   | 31       | 0.67              | 19   | 93       | 0.20                   | 164   |

<table>
<thead>
<tr>
<th>Codebase</th>
<th>Inspections</th>
<th>$ro$’s</th>
<th>$\neg ro$’s</th>
<th>$co$’s</th>
<th>$\neg co$’s</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDL</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>OpenSSH</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>19</td>
<td>1</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>GIMP</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>XNU</td>
<td>10</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>16</td>
<td>4</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>Linux</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>17</td>
<td>3</td>
<td>18</td>
<td>2</td>
</tr>
</tbody>
</table>
Evaluation

Three AFGs

- Basic AFG - as described earlier
- AFG NoFPP - AFG that does no false path pruning
- AFG Rename - Renames functions at each callsite to make all function calls unique

![Graphs showing performance comparison between AFG, AFG-NoFPP, and AFG-Rename for two different conditions: (a) SDL: ro, (b) SDL: co.](image)
(i) Linux: $ro$

(j) Linux: $co$
Take away

- Technique works assuming assumption about programming idioms hold
- This is often true except in special cases such as the Linux kernel