Static Detection of Dynamic Memory Errors

Written by David Evans

SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’96) – Philadelphia May 1996

Topics Covered

- The Problem
- Proposed Solution
- LCLint Checking Tool
- Analysis
- Toy Database Example
- Highlights
- Discussion

The Problem

Invalid assumptions in software lead to ineffective detection of bugs at compile time

Proposed Solution

Combination of two techniques

- **Annotations**
- **Static checking tool**

LCLint Checking Tool

- Introduced by David Evans in 1994 [1,2]
- Checks procedures independently
- Extended to include broad class of important errors
  - Misuse of null pointers
  - Memory usage
  - Storage
  - Aliasing
- Likely-case assumptions (average-case)

Why Enhance LCLint?

- LCLint
  - Over 100K lines of source code
  - Developed by 3 “different” authors
  - Did not attempt to allocate memory completely
  - Garbage collector
  - Limited portability

An Enhanced LCLint

- Explicit memory deallocation
- Memory annotations
- Libraries to store interface information
  - Increased performance
- Relaxed checking

Previous Attempts

Academic and commercial projects

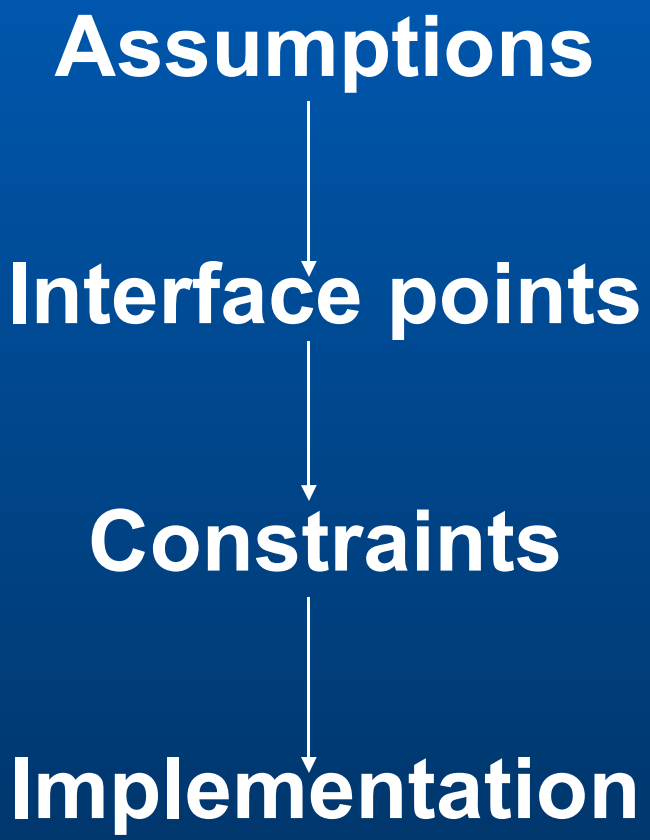
dmalloc, mprof, and Purify [Pure,Inc.]

- **Localize the symptom of a bug**
- **Sometimes require new search**

To Annotate or Not?

- Rules can be used to combine values from assumptions at confluence points
- Easier to write than LCL function specifications
- Source code can be written to annotate manually
  - /*@[annotation]@*/

Annotations



How Is This Useful? Storage

Storage model

- **Keywords**
  - **object**
  - **undefined**
  - **defined**
  - **lvalue**
  - **pointer**

Null Pointer Example

```
extern char *gname;

void setName (char *pname)
{

        gname = pname;

}
```

Null Annotation

```
extern char *gname;

void setName (/*@null@*/ char *pname)
{
        gname = pname;

}
```

Truenull annotation

```
extern char *gname;
extern /*@truenull@*/
    isNull (/*@null@*/ char *x):

void setName (/*@null@*/ char *pname)
{
    if (!isNull(pname))
            {
                    gname = pname;
            }
}
```

Definition

- LCLInt reports errors on “incomplete definitions”
  - Function parameters
  - Global variables used by a function

Allocation

- **Deallocation Errors**
  - **Live references to same storage**
  - **Failing to deallocate storage before the last reference to it is lost**

Storage Release Obligation

- ***only* annotation**
  - **Reference is the only pointer to the object it points to**
    - **Pass as an parameter corresponding to formal parameter with the only notation**
    - **Assign an external reference declared with an *only* annotation**
    - **Return it as a result declared with an *only* annotation**
  - **After release obligation is transferred, the original reference is a dead pointer and the storage it points to may not be used**

Aliasing

Unexpected aliasing

- Deallocation errors

returned

- Return value may alias the parameter

unique **(similar to *only*)**

- Does not transfer obligation to release storage
- Does not prevent aliasing that is invisible to the called function

Example

- "Toy" employee database program
  - 1000 lines of source code
  - 300 lines of interface specifications
  - Correction of original program with "older" LCLint
  - Interpretations of declarations
    - no null pointer annotations
    - No definition annotations

What Were the Results?

- Potential dereferencing of null pointers
- Instances where obligation to release storage was not transferred to the caller
- Assignment of allocated storage to fields of a static variable
- Six memory leaks reported
- Identifies where variables should not share any storage

Drawbacks

- Disadvantages
  - Execution flow analysis
  - Explicit de-allocation yields more bugs
  - Missing annotations in the standard library
  - Free global storage before execution terminates
  - No experience implementing as a new program is developed

Highlights

- Promising for memory allocation
  - Ad hoc and run-time testing methods failed
- Improve program documentation
- Combination of static, run-time, and extensive proves useful

Discussion

- What other key problem errors in software development would be great targets for implementing “annotations”?
- How does this compare to the Daikon-processing of invariants? Time? Thought process?
- How granular does a specification need to be in order for this to work?
- Is there a formula to determine the number of test cases or times required to run LCLint to get the best results?

References

[1] David Evans, *Using Specifications to Check Source Code*, MIT/LCS/TR-628, MIT Laboratory for Computer Science, June 1994

[2] David Evans, John Guttag, Jim Horning, and Yang Meng Tan. *LCLInt: A tool for using specifications to check code,* SIGSOFT Symposium on the Foundations of Software Engineering, December 1994