LCLint

Lecture 3
Outline

• LCLint
  - History, philosophy
  - Basic checking

• Limitations

• New & Improved LCLint
The Idea

• C programs have lots of bugs
  - Due to weaknesses of language
  - Emphasis on performance over safety
  - Era in which C was born

• Today we could design a much better C
  - But replacing C would be hard

• Retrofit this knowledge in a C tool
Philosophy

• Easy to learn

• Incremental benefit for incremental effort
  - Some benefit with zero effort
  - More specifications, more checking

• Efficiency
  - No overnight analysis, please

• Flexible
  - Flag city
Another take on Philosophy

- Larch is a major specification/theorem proving project at MIT
  - Very long-lived

- LCLint was born from Larch
  - Tries to address perceived problems with Larch
Features

• **Check abstraction boundaries**
  - E.g., direct client access to representations
  - Requires programmer annotations

• **Undocumented**
  - Use of globals
  - Modification of externally visible state

• **Missing initialization**
Basics

• LCLint adds a bool type to C
  - And understands that type

• Checks that predicates have type bool
  - With appropriate flag settings
  - Catches the classic

    if (x = y) ...

• This is fixed in Java
Expressing Abstraction

• Rewrite modules into three files
  - Module.c the code, as usual
  - Module.h “private” header
  - Module.lcl “public” header

• The .lcl file contains external interface
  - Function prototypes, global variables, etc.
Checking Abstraction

• Abstraction is enforced via visibility rules
  - Within a module, the representation is visible
  - Outside a module, only the external interface is visible

• Thus, checking abstraction boils down to type checking
  - Just as in Java, C++
Checking State Changes

• LCLint provides **modifies** clauses for declaring allowable updates to global state

```c
void copyDate (date *d1, date *d2)
    {modifies *d1;}
```

• Simply says that `copyDate` may modify its first argument
  - Doesn’t fully handle aliasing, though
Out Parameters

- **C is weak on function results**
  - Return value often needed for error code

- **Idiom: One of the arguments is passed only to hold the result**

- **Declare explicitly with `out` declaration**
  - `out` parameters should not be read
Summary

• Encode properties as types

• Reduce problems to type checking

• For efficiency, require sufficient information on functions to typecheck body in isolation
  - Forces annotations on function prototypes
  - No support for type inference
Weaknesses

• LCLint v1.0 is *flow insensitive*

• Types cannot change
  - The type of a value is permanent
  - The same for the entire scope of the variable

• Thus, LCLint cannot check flow sensitive properties
A Flow-Sensitive Property

Is a pointer null?

```c
char *x = malloc( . . .)    \[x \text{ may or may not be null}\]
if (x)
    \{ \[x \text{ definitely not null}\]
else
    \{ \[x \text{ definitely null}\]
```

Note: x’s type is flow insensitive, its nullness if flow sensitive
Analyzing Memory

• LCLint was extended to analyze memory usage

• Motivated in part by the poor memory management in LCLint
  – And failed attempts to fix it

... its implementation with regard to memory management is horrible. Memory is allocated willy-nilly without any way to track it or recover it. Malloced pointers are passed and assigned in a labyrinth of complex internal data structures. ...
Analyzing Memory (Cont.)

- Memory goes through many stages:
  - Allocated
  - Assigned
  - Read
  - DEALLOCATED

- There are implicit safety rules
  - E.g., no read after deallocation

- These are flow sensitive properties
  - Suggests dataflow analysis
Remember Available Expressions?

if y > a + b

a := a + 1

x := a + b

y := a * b

a + b

a + b, a * b

a + b

a + b

a + b, y > a + b
Framework

• **Goal:** Preserve local checking
  - Annotate functions with sufficient information

• **Example:**

```c
extern char *gname;

void setName (char *pname) {
    gname = pname;
}
```
Questions

extern char *gname;

void setName (char *pname) {
    gname = pname;
}

• Can `pname` be null?
• Was `gname` the sole reference to storage?
• Does the caller deallocate `pname`?
Annotations: Only

• Only storage declares a unique reference to storage

```c
extern only char *gname;

void setName (char *pname) {
    gname = pname;
}
```

• Error: unique reference is lost
Only (Cont.)

- **Only** references cannot be lost
  - But they can be transferred

- **Consider the signature of** `free`
  `void free (only void *ptr)`

- **Now**
  
  ```c
  { x is only here } 
  free(x) 
  {x is marked as inaccessible here}
  ```

  - Note the flow sensitivity!
Computing Flow-Sensitive Information

• Flow-sensitive information can be expensive
  - Folk wisdom: interprocedural analysis too expensive

• LCLint analyzes each function body separately
  - All needed information must be declared at function interfaces

• LCLint properties are atomic
  - null, not-null, only, temp, returned

• Flow-sensitive analysis of atomic properties in a single procedure is dataflow analysis
Consider:
```
extern null char *gname;
```

- `gname` is declared possibly null
  - Any use must be guarded by a test
  - LCLint must be able to analyze predicates
    - Recognize `== NULL, != NULL`
    - Annotations `truenull, falsenull` for function calls
    - This is a more complex flow sensitive analysis
Alternatively:

```
extern char *gname;

... = *gname;
```

- **gname** is declared as never null
  - No need for tests
  - But
    - Cannot be assigned the value of a declared null pointer
    - Cannot be assigned NULL
Null

• For each variable, track:
  - null, not-null, maybe null
  - Must also track fields of structures
  - LCLint provides annotations to support this
    • E.g., Fields can be declared as `null`

```c
{gname may be null}
if (gname)
  { gname not null}
  ... = *gname;
```

• Forward, may analysis
• Terminates
  - Domain is finite
Aliasing

• LCLint provides support for detecting aliases
  - Nearly unique in this respect
  - Many tools ignore aliasing

• Examples:
  - `foo(returned char *x)`
    • Return value of `foo` may alias `x`
      - For tracking aliases across function calls
  - `foo(temp char *x)`
    • No new, visible aliases of `x` may be created
Aliasing

• For each variable, keep track of possible aliases

• Example:

  l = x;
  { l aliases x}
  if (...) l = l->next; {l aliases x->next}
  {l may alias x, x->next}

• Forward, may alias analysis

• But domain is not finite!
  - Guarantee termination by ignoring loops - unsound yet useful
Conclusion

• LCLint is ad hoc in many ways
  - Unsound
  - Rough treatment of loops
  - Annotations are a mish mash of ideas

• But, a success story
  - Lots of ideas
  - Fairly widely used