Nomenclature

- **Static**
  - Anything algorithm that analyzes source code only
  - Does not run the program
- **Bug Detection**
  - Finding some bugs
  - No promise made that every bug is found
    - False negatives
  - No promise made that every bug report is a bug
    - False positives

This Lecture

- **Lint**
- **LCLint**

C Lint

- Modern C compilers check for many questionable constructs:
  
  ```c
  if (a == b) . . .
  should probably be  a == b
  
  if (a == 1 && b == 2) . . .
  should probably be  a == 1 & b == 2
  
  if (x << y + 2) . . .
  parentheses preferred  x << (y + 2)
  ```

C Lint (Cont.)

- Where did this idea come from?
- **C Lint**
  - Stand-alone program
  - Checked for many such idiomatic errors
  - Written in 1979

LCLint

- History, philosophy
- Basic checking
- Limitations
- New & Improved LCLint
  - Interlude: Dataflow analysis
The (Same) 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
    \[ \text{if } (x = y) \ldots \]
- 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
  
  ```
  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 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?

```
char *x = malloc(...)
x may or may not be null
if (x)
x definitely not null
else
x definitely null
```

Note: x’s type is flow insensitive, its nullness is 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

Framework

- Goal: Preserve local checking
  - Annotate functions with sufficient information
- Example:
  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
  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
  (x is only here)
  free(x)
  (x is marked as inaccessible here)
- Note the flow sensitivity!
Consider:

```c
extern null char *gname;
if (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:

```c
extern char *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

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

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

Interlude: A Short Primer on Dataflow Analysis

Control-Flow Graphs

```
x := a + b;
y := a * b;
while y > a + b {
  a := a + 1;
x := a + b
}
```

Control-flow graphs are state-transition systems.
Notation

- \( s \) is a statement
- \( \text{succ}(s) = \{ \text{successor statements of } s \} \)
- \( \text{pred}(s) = \{ \text{predecessor statements of } s \} \)
- \( \text{write}(s) = \{ \text{variables written by } s \} \)
- \( \text{read}(s) = \{ \text{variables read by } s \} \)

Note: In literature write = kill and read = gen

Available Expressions

- For each program point \( p \), which expressions must have already been computed, and not later modified, on all paths to \( p \).
- Optimization: Where available, expressions need not be recomputed.

Dataflow Equations

\[
\begin{align*}
A_0(s) &= \begin{cases} 
\emptyset & \text{if } \text{pred}(s) = \emptyset \\
\bigcup_{i \in \text{pred}(s)} A_i(s') & \text{otherwise}
\end{cases} \\
A_{\infty}(s) &= (A_0(s) - \{a \in S | \text{write}(s) \cap \text{V}(a) = \emptyset\}) \\
& \cup \{s | \text{write}(s) \cap \text{read}(s) = \emptyset\}
\end{align*}
\]

Example

\[
\begin{align*}
x := a + b \\
y := a + b \\
a := a + 1 \\
x := a + b \\
\end{align*}
\]

Liveness Analysis

- For each program point \( p \), which of the variables defined at that point are used on some execution path?
- Optimization: If a variable is not live, no need to keep it in a register.

Dataflow Equations

\[
L_\infty(s) = (L_{\text{out}}(s) - \text{write}(s)) \cup \text{read}(s)
\]

\[
L_{\text{out}}(s) = \begin{cases} 
\emptyset & \text{if } \text{succ}(s) = \emptyset \\
\bigcup_{s' \in \text{succ}(s)} L_\infty(s') & \text{otherwise}
\end{cases}
\]
Example

Available Expressions Again

Available Expressions: Schematic

Live Variables Again

Live Variables: Schematic

Solving Dataflow Equations

• Simple worklist algorithm:
  - Initially let $S(v) = 0$ for all $v$
  - Repeat until $S(v) = S(E)$ for all equations
    • Pick any $v = E$ such that $S(v) \not\subseteq S(E)$
    • Set $S := S(v)/S(E)$
Termination

- How do we know the algorithm terminates?

- Because
  - the sets only get bigger (or smaller) on each iteration
  - the domain is finite

Finale: LCLint Dataflow Analysis

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

Aliasing

- For each variable, keep track of possible aliases

  - Example:
    - \( I = x; \)
    - \( \{ I \text{ aliases } x \} \)
    - \( \text{if } (...) I = l->\text{next}; \{ I \text{ aliases } x->\text{next} \} \)
    - \( I \text{ may alias } x, x->\text{next} \)

  - Forward, may alias analysis
  - But domain is not finite!
    - Guarantee termination by ignoring loops

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