TDDE45 - Lecture 6: Metaprogramming and Debugging

Adrian Pop and Martin Sjölund

Department of Computer and Information Science
Linköping University

2023-09-27

Part I

Metaprogramming

What is metaprogramming?

> *Meta (from the Greek meta- meaning "after" or "beyond") is a prefix used in English to indicate a concept that is an abstraction behind another concept, used to complete or add to the latter.*
>
> *Wikipedia*

Metaprogramming is a higher order programming, often described as a program where the data is source code.
With this definition, every compiler is a metaprogram.

Different kinds of metaprogramming

These are the main types of metaprogramming:

- Generative programming
- Template programming
- Reflective programming
- Macro programming

Generative programming

A program which generates other programs. Many domain-specific languages fall into this category:

- Graphviz (generates postscript, etc)
- Embedded Lua (the main program can generate Lua programs at runtime)
- cmake/autoconf (generates Makefiles)
- C preprocessor (generates C-files)
- Any source-to-source compiler
- And many more...

Macro programming

A macro takes as input code and transforms it into different code in the same language. Different languages have different levels of power that the macro language allows.

- C preprocessor macros are simple text substitution on tokens (and a few simple conditional checks).
- autoconf (in the build systems part of the last lab) uses m4 as its macro processor, which is more powerful than C preprocessor.
- Julia macros on the other hand allow virtually any function to be called from a macro.

Note that you are not always guaranteed that the output will be syntactically valid.

Macro programming – C preprocessor

```
#define MIN(X,Y) ((X) < (Y) ? (Y) : (X))

MIN(2+4/7,3)
// generates
(2+4/7) < (3) ? (2+4/7) : (3)

MIN(sin(y),3)
// generates code where sin(y) is called twice
(sin(y)) < (3) (sin(y)) : (3)

// Also note that the parenthesis are needed
#define CUBE(I) (I * I * I)
CUBE(1+1)
// generates
(1+1 * 1+1 * 1+1)
```

Macro programming – Lisp Simple Example

```
; In lisp:
; a macro returns an expression
(defmacro my-double (x) `(* 2 ,x))
MY-DOUBLE

(macroexpand-1 `(my-double 3))
(* 2 3)

(my-double 3)
6
```

```
; Note that you cannot use:
(defmacro my-double-fail (x) (* 2 x))
MY-DOUBLE-FAIL

(my-double-fail 3)
6

(let ((y 3)) (my-double-fail y))
; caught ERROR:
;   during macroexpansion ...
;
;    The value
;      Y
;    is not of type
;      NUMBER
```

Macro programming – Lisp Example Usage

```
(macroexpand `(dolist (i (list 'a 'b 'c)) (print i)))

(BLOCK NIL
  (LET ((#:N-LIST392 '(A B C)))
    (TAGBODY
     #:START393
      (UNLESS (ENDP #:N-LIST392)
        (LET ((I (CAR #:N-LIST392)))
          (SETQ #:N-LIST392 (CDR #:N-LIST392))
          (TAGBODY (PRINT I)))
        (GO #:START393))))
  NIL)
```

Template programming

Less powerful than a full macro system which can manipulate entire ASTs.
The basic idea is that you define a template as a class, function, etc that contain some blanks that can be filled in.
When instantiating the template with a particular argument, a concrete class or function is generated from this.

Template programming - Defining a template

```
template<typename T1, typename T2>
struct Pair {
  T1 first;
  T2 second;
};
```

Template programming - Instantiating a template

```
template<typename T1, typename T2>
struct Pair {
  T1 first;
  T2 second;
};

const Pair<int,double> p = {.first = 1, .second = 2.0};

// Generated by the above (also possible to define explicitly)
template<> struct Pair<int,double> {
  int first;
  double second;
};
```

C++ template programming is Turing-complete

```
template <int N> struct Factorial
{
    enum { val = Factorial<N-1>::val * N };
};

template<>
struct Factorial<0>
{
    enum { val = 1 };
};

const int n4 = Factorial<4>::val;
```

Runtime performance of template programming

Templates can make it possible specialize code, making the compiler optimize it automatically:

```
template <int length>
Vector<length>& Vector<length>::operator+=(const Vector<length>& rhs)
{
    for (int i = 0; i < length; ++i)
        value[i] += rhs.value[i];
    return *this;
}

template <>
Vector<2>& Vector<2>::operator+=(const Vector<2>& rhs)
{
    value[0] += rhs.value[0];
    value[1] += rhs.value[1];
    return *this;
}
```

Example from `https://en.wikipedia.org/wiki/Template_metaprogramming`

Disadvantages of template programming

Templates can make it possible specialize code, making the compiler optimize it automatically.

This generates many similar functions or classes. It can take a lot of time and space to optimize and generate all of the code.

Note that Java generics is similar to C++ templates, but does not have a compile-time penalty (and will not specialize the function in the same way C++ templates does).

Reflective programming

The ability to examine, introspect, and modify your own structure and behavior.
The next few slides have some examples from:
`https://en.wikipedia.org/wiki/Reflection_(computer_programming)`

Reflective programming in C#

```
// Without reflection
Foo foo = new Foo();
foo.PrintHello();

// With reflection
Object foo = Activator.CreateInstance("complete.classpath.and.Foo");
MethodInfo method = foo.GetType().GetMethod("PrintHello");
method.Invoke(foo, null);
```

Source: Wikipedia

Reflective programming in Java

```java
import java.lang.reflect.Method;
// Without reflection
Foo foo = new Foo();
foo.hello();
// With reflection
try{
  // Alternatively: Object foo = Foo.class.newInstance();
  Object foo = Class.forName("path.to.Foo").newInstance();
  Method m = foo.getClass().getDeclaredMethod("hello", new Class<?>[0]);
  m.invoke(foo);
} catch (Exception e) {
  // Catching ClassNotFoundException, NoSuchMethodException
  // InstantiationException, IllegalAccessException
}
```

Source: Wikipedia

Reflective programming in Python

```python
# without reflection
obj = Foo()
obj.hello()

# with reflection
obj = globals()['Foo']()
getattr(obj, 'hello')()

# with eval
eval('Foo().hello()')
```

Source: Wikipedia

What use is there for reflective programming?

- The ability to inspect its own members allows you to (de-)serialize data corresponding to the object.
- Finding all classes that implement a specific interface allows you to create plugins. (Those who did the build systems lab used dlopen, which contains part of what reflection is)
- Software testing.
- Some design frameworks like to use reflective programming.
- Useful to create debugger and other tools based on your language.
- Accessing classes as data types is fun.

Part II

Debugging

Avoid writing “clever” code

> *Everyone knows that debugging is twice as hard as writing a program in the first place.*
> *So if you’re as clever as you can be when you write it, how will you ever debug it?*
>
> *The Elements of Programming Style, Kernighan and Plauger 1978*

Testing

Debugging and testing are very much related.
Many testing techniques are applicable to debugging:

- static code analysis
- memory checkers
- code coverage checkers
- fuzz testing

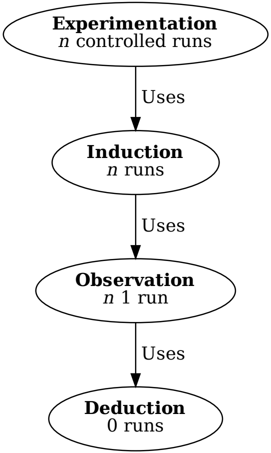
If you are good at debugging code, you are probably proficient at testing code as well.
There are of course differences.

Differences with testing

- Are you looking to fix a particular bug (debugging) or looking to find bugs (testing)?
- Testing is often automated; but debugging usually cannot be automated.
- Debugging requires you to understand the bug.

Reasoning about programs

- Experimentation – refinining and rejecting a hypothesis
- Induction – the program crashes for some inputs in the test suite
- Observation – the program always crashes
- Deduction (static analysis)



Based on Zeller 2009, p.142 (Figure 6.5).

Static code analysis

Examining the code without running the program.

- Possible to use during the testing phase.
- Usually focused on detecting some code smells or common errors
  - Unused variables
  - Read from uninitialized variables
  - Unreachable code
  - Dereferencing null pointers
  - Misusing an interface (usually only standard library)
- False positives
- Halting problem

List of tools available here:
`https://github.com/analysis-tools-dev/static-analysis`

Static code analysis – ccc-analyzer (LLVM project)

Bug Summary

File: scanbuild.c
Warning: line 9, column 3
1st function call argument is an uninitialized value

Report Bug

Annotated Source Code

[?]

```
1  #include <string.h>
2  #include <stdlib.h>
3
4  int main(int argc, char **argv) {
5    double *d;
```

1 'd' declared without an initial value →

```
6    if (argc > 1) {
```

2 ← Assuming 'argc' is <= 1 →

3 ← Taking false branch →

```
7      d = malloc(sizeof(double)*2);
8    }
9    bzero(d, sizeof(double)*2);
```

4 ← 1st function call argument is an uninitialized value

```
10   return 0;
11 }
```

Dynamic analysis

Any kind of analysis performed on a running program.

- Debugger
- Simulator
- Various traces

Dynamic analysis – debugger (GDB)

A debugger is what people think of when you say you are debugging a program.

- Fast. There is usually hardware support available.
- Attach to a running process (to see where it got stuck).
- Inspects the runtime stack to get information about values of local variables, names of functions, etc.
- Ability to set breakpoints and watchpoints.
- Less information if compiled without debug symbols (-g), but still works.
- Note that the debugger often zeros out memory automatically, hiding some bugs when the program is executed by the debugger.
- Command-line GDB is a useful tool to learn for debugging on servers, etc.
- Graphical frontends such as ddd, gdbgui, Eclipse, VSCode, etc.

Dynamic analysis – valgrind

One advantage of valgrind (Nethercote and Seward 2007) is that you typically do not need to recompile programs in order to run it (although compiling with `-O1 -g` is recommended to get better performance and locations of errors).

Valgrind is a suite of dynamic debugger tools. The tools will interpret your program and keep track of memory allocations or CPU consumption, etc. Typical slowdown is 10-100x (depending on which tool is used).

The memory checker (default tool) will detect:

- Read access to uninitialized memory
- Write or read access to (nonallocated memory / free'd memory / certain stack areas)
- Memory leaks (memory that is not free'd when the program exits)

Dynamic analysis – what valgrind (memcheck) cannot find

```c
#include <string.h>
#include <assert.h>
#include <stdlib.h>
struct s {
  double *d;
  int i;
};
int main() {
  struct s my_s = {.d = malloc(sizeof(double)*2), .i = 1};
  bzero(&my_s.d, sizeof(double)*2);
  assert(my_s.i == 1);
  return 0;
}
```

Fails to give the cause of assertion failure.

Cannot replicate the bug?

- Undefined behaviour.
- Random behaviour, context switches, etc. Determinstic?
- Operating environment (different hardware, OS, installed packages, etc)
- Only occurs in certain situations?
  - Does your code work when the client is behind the Great Firewall of China, on a cell phone with 2G connection or only on the developer's LAN connection?
- What version of the software triggered the bug?
- Bad bug report? Can the user still replicate the bug?

Cannot find the cause?

You have a known bad state in the program that you can identify – try to go backwards. Modify your program if needed.

- Add assertions at appropriate places – values you "know" to be non-NULL for example.
- Add other sanity checks (expensive ones) that can be turned off easily.
- Tracing, logging, printf statements – search for a location of a good state close to the bad state. Set a debugger breakpoint or here and step through single instructions or set a watchpoint.
- `strace` – see all system calls performed by the program in a sequence. This can sometimes help figure out what is going wrong. File not found? Out of file descriptors?
- Create more unit tests in the affected code.
- Run code coverage tools while you are triggering the bug. Does it run the expected code paths?

Where is the bug documented?

You need a bug tracking system for your software.

- Which bugs were solved since the latest release?
  - How was it resolved? In which commit? Chances are similar issues will pop up later.
- Which bugs are still open?
- Always add your bugs to the bug tracker.
  - Also add which approaches you tried for replicating the bug so you (or someone else) does not have to redo this work when trying to resolve the bug later on.

Profiling

- System runs out of memory when your program runs?
- Program worked fine for small inputs but starts using lots of processing power for medium-size inputs?
- Profiling tools helps you solve problems with resources.
  - See on which lines memory is allocated.
  - Which functions take the longest time to execute.

> “*Premature optimization is the root of all evil.*
>
> *Donald Knuth*”

CPU Profiling Techniques

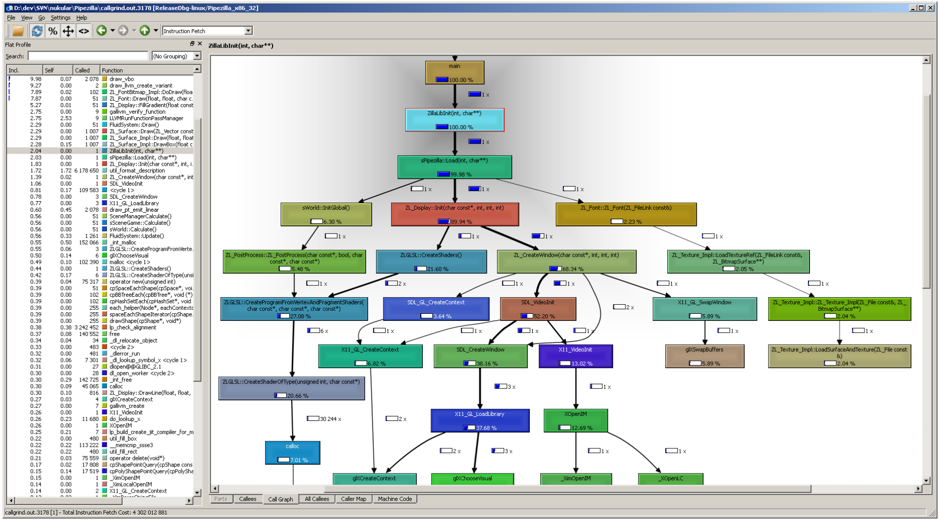
- Statistical profiling (poll the program regularly)
  - Code instrumentation (`-pg` flag for gprof)
  - Kernel support (`operf` in Linux)
  - Basically using a debugger (stop at random intervals checking the stack frames)
- Interpret or simulate the program (including cache hits or misses)

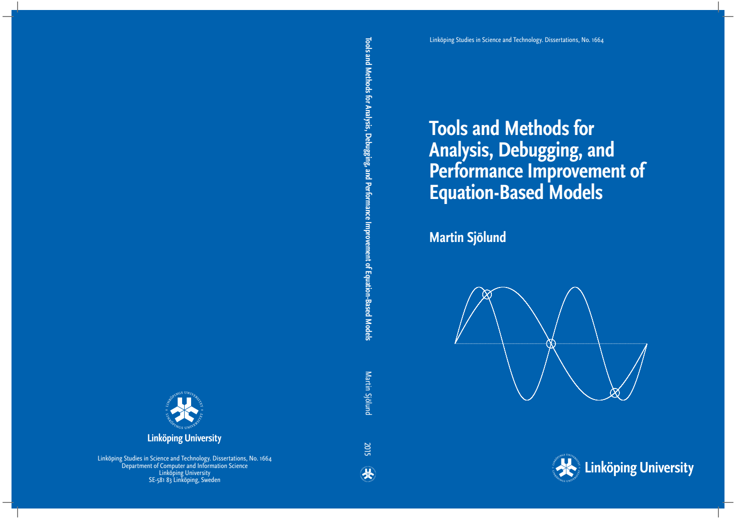
CPU Profiling Techniques – operf

Requires root user or kernel flags to be enabled/disabled.

```
$ oprof omc hello.mos # Load the Modelica Standard Library
$ opreport
 ...
  samples|      %|
------------------
   152021 100.000 operf
  cpu_clk_unhalt...|
    samples|      %|
  ------------------
      65841 43.3105 libomantlr3.so
      29943 19.6966 libomcgc.so.1.4.1
      24493 16.1116 libOpenModelicaCompiler.so
      21703 14.2763 kallsyms
       7728  5.0835 libc-2.27.so
       1075  0.7071 libpthread-2.27.so
        490  0.3223 libomcruntime.so
        337  0.2217 ld-2.27.so
        318  0.2092 libopenblasp-r0.2.20.so
```

Valgrind - callgrind & kcachegrind





Martin's dissertation – focused on debugging tools for equation-based models (where the control flow is not dictated by the program).

Resources

- See how some bug reports look like for OpenModelica: `https://github.com/OpenModelica/OpenModelica/issues`
- See how some Pull Requests (PRs) look like for OpenModelica: `https://github.com/OpenModelica/OpenModelica/pulls`
- Valgrind tools:
  - `https://valgrind.org/docs/manual/mc-manual.html`
  - `https://valgrind.org/docs/manual/cl-manual.html`
  - `https://valgrind.org/docs/manual/hg-manual.html`

References

[KP78] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style*. 2nd. McGraw-Hill, 1978. ISBN: 0070342075.

[NS07] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '07. San Diego, California, USA, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746.

[Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. 2nd ed. Morgan Kaufmann Publishers Inc., 2009. ISBN: 978-0-12-374515-6.

www.liu.se