Towards a dynamic object model within Unix processes

Stephen Kell
stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge
Fragmentation by VM
Language VMs have failed.

- none has won
- haven’t dislodged Unix-like abstractions
- fragmentation, complexity

Try

- extending Unix…
- … so that it *embraces* and *integrates* VMs!
- hypothesis: possible without throwing away VMs
Some things missing from Unix:

- fine-grained *object-like* notion
- semantic metadata ("type info")
- efficient binding from object to metadata
Some things missing from Unix:

- fine-grained *object-like* notion
- semantic metadata (“type info”)
- efficient binding from object to metadata

... so I’ve built **liballocs**, which adds them!

- core abstraction: *allocations*
- implements ≈ a meta-object protocol *process-wide*
- “type” metadata
- based on existing VM-like *rudiments* within Unix
- → very compatible, very general
Application 1: reflective checks in native code

For example...

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker, (struct commit *)obj))
        return -1;
    return 0;
}
```
Application 1: reflective checks in native code

For example...

```c
if (obj->type == OBJ_COMMIT) {
    if (process_commit(walker,
        (assert(_is_a(obj, &_uniqtype_commit)),
        (struct commit *)obj)))
        return -1;
    return 0;
}
```
Application 2: FFI-less scripting

$ ./node  # ←  a popular JavaScript implementation
Application 2: FFI-less scripting

$ ./node  # <—- ... with liballocs extensions
> process.lm.printf ("Hello, world!\n")
Hello, world!
14
Application 2: FFI-less scripting

$ ./ node # with liballocs extensions
> process.lm.printf ("Hello, world!
")
Hello, world!
14
> require('−lXt ');

Application 2: FFI-less scripting

$ ./node # with liballocs extensions
> process.lm.printf("Hello, world!
")
Hello, world!
14
> require('−lXt ')
> var toplvl = process.lm. XtInitialize ( 
    process.argv[0], "simple", null, 0,
    [process.argv.length], process.argv);
var cmd = process.lm.XtCreateManagedWidget( 
    "exit ", commandWidgetClass, toplvl, null, 0);
process.lm.XtAddCallback(
    cmd, XtNcallback, process.lm.exit, null );
process.lm.XtRealizeWidget(toplvl);
process.lm.XtMainLoop();

Not “JS ↔ C”! One object space, many per-language views
Application 3: precise debugging

(gdb) print obj
$1 = (const void *) 0x6b4880 # unknown type!
Application 3: precise debugging

(gdb) print obj
$1 = (const void *) 0x6b4880 # unknown type!

(gdb) print __liballocs_get_alloc_type (obj)
$2 = (struct uniqtype *) 0x2b3aac997630
   <__uniqtype__InputParameters>
Application 3: precise debugging

(gdb) print obj
$1 = (void *) 0x6b4880
(gdb) print __liballocs_get_alloc_type (obj)
$2 = (struct uniqtype *) 0x2b3aac997630
   <__uniqtype__InputParameters>
(gdb) print *(struct InputParameters *) $2
$3 = {ProfileIDC = 0, LevelIDC = 0, no_frames = 0,
       ... }

Better debugger integration to follow...
m = mmap(NULL, sz, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, 0);

Files are opaque bytes...
Application 4: sane approaches to file I/O

\[ m = \text{fmap}(\text{NULL}, \text{sz}, \text{PROT\_READ}\mid\text{PROT\_WRITE}, \text{MAP\_PRIVATE}, \text{fd}, 0, \&\text{uniqtype}\_\text{git\_cache}) \quad // \quad \text{describes an on-disk format} \]

Files are opaque bytes \emph{heaps of typed allocations}
VM-like rudiments

- dynamic loading
- dynamic (re)compilation
- dynamic binding
- reflection
- garbage collection

*Modern* Unix already has these things! . . . sort of.
// "forward" lookup, by name

double(*p_ceil)(double)
  = dlsym(RTLD_DEFAULT, "ceil");
// "forward" lookup, by name

double(*p.ceil)(double)
  = dlsym(RTLD_DEFAULT, "ceil");

// "reverse" lookup, by address

DL_info i;
dladdr(p.ceil, &i);
printf("%s\n", i.dli_sname); // "ceil"

// but only for "static" objects (known to loader)
// ... not stack, heap, etc..
$ cc -g -o hello hello.c && gdb -q --args ./hello
Reading symbols from /tmp/hello...done.
(gdb)
$ cc -g -o hello hello.c && gdb -q --args ./hello
Reading symbols from /tmp/hello...done.
(gdb) break main
Breakpoint 1 at 0x40053a: file hello.c, line 5.
(gdb) run
Starting program: /tmp/hello

Breakpoint 1, main () at hello.c:5
5 printf("Hello, world!\n");
We identify three design principles for reflection and metaprogramming facilities in object-oriented programming languages.

1. INTRODUCTION

Object-oriented languages traditionally support meta-level operations such as introspection, reflection, and metaprogramming. In a typical object-oriented language with reflection (e.g., Java, C#, Smalltalk), one might query an instance for its class, as indicated in the pseudo-code below:

```java
class Car {...} 
Car myCar = new Car(); 
int numberOfDoors = myCar.numberOfDoors(); 
Class theCarsClass = myCar.getClass(); 
Car anotherCar = theCarsClass.newInstance(); 
Class theCarsSuperclass = theCarsClass.getSuperclass();
```

Looking at the APIs of such a system, we expect to see something like:

```java
class Object { 
Class getClass(); 
...
}
class Class { 
Class getSuperclass(); // many other methods: getMethods(), getFields() etc. 
}
```

The APIs above support reflection at the core of the system. Every object has at least one reflective method, which ties it to `Class` and (most likely) an entire reflective system. Base- and meta-developers coexist side-by-side. The same class object ... The same object that exhibits behavior about the problem domain also exhibits behavior about being a member of a class (`getClass`).

This paper argues that meta-level functionality should be implemented separately from base-level functionality, using objects known as `mirrors`. Such an API might look something like this:

```java
class Object { 
// no reflective methods... 
} 
class Class { 
// no reflective methods... 
} 
interface Mirror { 
String ... 
} 
interface ClassMirror extends Mirror { 
ClassMirror getClass(); 
... 
}
```

Unix debugging adopted the same principles long ago.
The liballocs meta-protocol

struct uniqtype;  /* type descriptor */
struct allocator;  /* heap, stack, static, etc */
uniqtype * alloc_get_type (void *obj);  /* what type? */
allocator * alloc_get_allocator (void *obj);  /* heap/stack? etc */
void * alloc_get_site (void *obj);  /* where allocated? */
void * alloc_get_base (void *obj);  /* base address? */
void * alloc_get_limit (void *obj);  /* end address? */
Dl_info alloc_dladdr (void *obj);  /* dladdr−like */

Not “for C”; for any language! This is the API in C.
struct ellipse {
    double maj, min;
    struct point { double x, y; } ctr;
};
Compatibility with existing language implementations

Two cases!

Unix-style compilation toolchains (C, C++, Fortran, ...)

- *augment* the toolchain
- mostly generic, + a little per-language effort
- mostly done, working (esp. for C)

Existing language VMs

- retrofit onto liballocs APIs
- hypothesis: small changes only
- mostly future work
source tree

main.f  widget.  util.c  ...

/bin/ foo
/lib/ libxyz. so

compile and link

/load, link and run (ld.so)

foo (process image)
source tree

main.f  widget. C  util.c  ...

compiler wrappers

dump allocation sites (dumpallocs)

compile and link

/bin/  /lib/  libxyz.  .dbg/
foo    libxyz.  foo
   so     so

postprocess

liballocs.  foo-meta  libxyz-meta
  so       .so       .so

load, link and run (ld.so)

foo (process image)
<table>
<thead>
<tr>
<th>bench</th>
<th>normal/s</th>
<th>liballocs/s</th>
<th>liballocs %</th>
<th>no-load</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>4.91</td>
<td>5.05</td>
<td>+2.9%</td>
<td>+1.6%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.985</td>
<td>1.85</td>
<td>+88 %</td>
<td>– %</td>
</tr>
<tr>
<td>gobmk</td>
<td>14.2</td>
<td>14.6</td>
<td>+2.8%</td>
<td>+0.7%</td>
</tr>
<tr>
<td>h264ref</td>
<td>10.1</td>
<td>10.6</td>
<td>+5.0%</td>
<td>+5.0%</td>
</tr>
<tr>
<td>hmmer</td>
<td>2.09</td>
<td>2.27</td>
<td>+8.6%</td>
<td>+6.7%</td>
</tr>
<tr>
<td>lbm</td>
<td>2.10</td>
<td>2.12</td>
<td>+0.9%</td>
<td>(−0.5%)</td>
</tr>
<tr>
<td>mcf</td>
<td>2.36</td>
<td>2.35</td>
<td>(−0.4%)</td>
<td>(−1.7%)</td>
</tr>
<tr>
<td>milc</td>
<td>8.54</td>
<td>8.29</td>
<td>(−3.0%)</td>
<td>+0.4%</td>
</tr>
<tr>
<td>perlbench</td>
<td>3.57</td>
<td>4.39</td>
<td>+23 %</td>
<td>+1.6%</td>
</tr>
<tr>
<td>sjeng</td>
<td>3.22</td>
<td>3.24</td>
<td>+0.6%</td>
<td>(−0.7%)</td>
</tr>
<tr>
<td>sphinx3</td>
<td>1.54</td>
<td>1.66</td>
<td>+7.7%</td>
<td>(−1.3%)</td>
</tr>
</tbody>
</table>
Retrofitting a VM: the shopping list

Generate uniqtypes

■ need not be 1:1 with “type” in the language

Implement liballocs meta-protocol

■ ... some pre-fab options available

Notify dynamic loader of JITted code

■ extra goodie: libdlbind can do this

Whole-process binding...

■ slow path: just use the meta-protocol

■ fast path: no change! i.e. affinity for own objects
Core runtime and toolchain extensions work well

- really!

Next step: actually retrofit one or more VMs

- whole-process reflection
- whole-process tools (debuggers, profilers, ...)
- interop without FFIs (improving the node use-case)

Code is here: https://github.com/stephenrkell

- please get in touch

Thanks for listening... questions?
The real difference between Unix and VMs

In Smalltalk’s “integrated environment”... there is little distinction between the compiler, interpreter, browser and debugger, [all of which] cooperate through shared data structures.... Pi is an isolated tool in a [Unix] “toolkit environment” [and] interacts with graphics, external data and other processes through explicit interfaces.

T.A. Cargill

Pi: a case study in object-oriented programming

OOPSLA ’86
- encapsulation
- stratification
- ontological correspondence

Story in brief: Unix debugging has all three...

... decades before Mirrors were articulated
```
$ cc -g -o hello hello.c && readelf -wi hello | column

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_language</td>
<td>1 (ANSI C)</td>
</tr>
<tr>
<td>AT_name</td>
<td>hello.c</td>
</tr>
<tr>
<td>AT_low_pc</td>
<td>0x4004f4</td>
</tr>
<tr>
<td>AT_high_pc</td>
<td>0x400514</td>
</tr>
<tr>
<td>AT_type</td>
<td>&lt;0x2af</td>
</tr>
<tr>
<td>AT_name</td>
<td>main</td>
</tr>
<tr>
<td>AT_low_pc</td>
<td>0x4004f4</td>
</tr>
<tr>
<td>AT_high_pc</td>
<td>0x400514</td>
</tr>
<tr>
<td>AT_type</td>
<td>&lt;0xc5</td>
</tr>
<tr>
<td>AT_name</td>
<td>int</td>
</tr>
<tr>
<td>AT_type</td>
<td>&lt;0x2b5</td>
</tr>
<tr>
<td>AT_location</td>
<td>fbreg - 20</td>
</tr>
<tr>
<td>AT_type</td>
<td>&lt;0x7ae</td>
</tr>
<tr>
<td>AT_name</td>
<td>char</td>
</tr>
</tbody>
</table>
```
Retrofitting dynamic compilation

```c
void *obj = dlopen("libmylib.so", RTLD_NOW);
void *def = dlsym(obj, "my_symbol");
```

What about dynamic compilers?

```c
void *obj = dlcreate("codeheap", RTLD_NOW);
char *myfunc = dlalloc(obj, sz, 0);
/* ... compile something, or generate interpreter stub */
void *def = dlbind(obj, myfunc, "my_symbol", STT_FUNC);
```

Now your VM’s definitions are visible to liballocs

- ... and to gdb et al! (need minor extensions)
Retrofitting binding: in brief (1)

cmp [ebx, <class offset>], <cached class>; test
jne <inline cache miss> ; miss? bail
mov eax, [ebx, <cached x offset>] ; hit
Retrofitting binding: in brief (2)

xor ebx, <allocator mask> ; get allocator
cmp ebx, <cached allocator prefix> ; test
jne <allocator miss> ; miss? bail
cmp [ebx, <class offset>], <cached class>; test class
jne <cached cache miss> ; miss? bail
mov eax, [ebx, <cached x offset>] ; hit