Towards a dynamic object model within Unix processes

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge


GALLEY EDITOR
SMALLTALK-80
INTRODUCTION TO VLSI SYSTEMS

Fragmentation by VM

photo: AngMoKio

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…

```
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...

```
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!\n")
Hello, world!
14
> require('-lXt');
```

Application 2: FFI-less scripting

```
$ ./node # with liballocs extensions
> process.lm.printf("Hello, world!\n")
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 $\leftrightarrow$ 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…

Application 4: sane approaches to file I/O

```
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 = fmap(NULL, sz, PROT_READ|PROT_WRITE,
        MAP_PRIVATE, fd, 0,
       &__uniqtype__git_cache  //  describes an on-disk format
     );
```

Files are ~~opaque bytes~~ *heaps of typed allocations*

VM-like rudiments

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

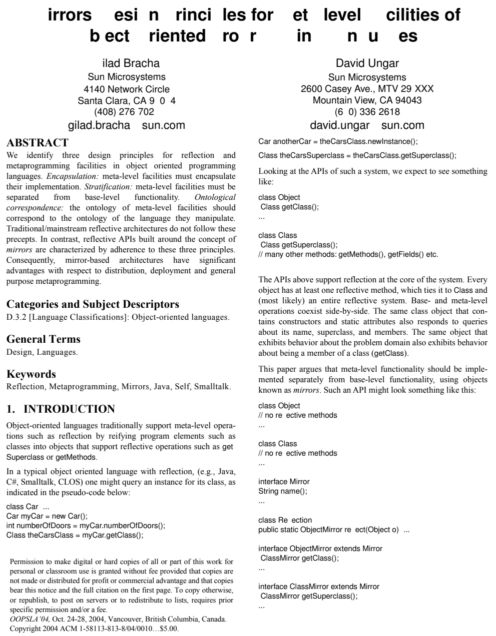
Modern Unix already has these things! ... sort of.

```
// "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");
```



irrors esi n rinci les for et level cilities of b ect riented ro r in n u es

ilad Bracha
Sun Microsystems
4140 Network Circle
Santa Clara, CA 9 0 4
(408) 276 702
gilad.bracha sun.com

David Ungar
Sun Microsystems
2600 Casey Ave., MTV 29 XXX
Mountain View, CA 94043
(6 0) 336 2618
david.ungar sun.com

ABSTRACT

We identify three design principles for reflection and metaprogramming facilities in object oriented programming languages. *Encapsulation:* meta-level facilities must encapsulate their implementation. *Stratification:* meta-level facilities must be separated from base-level functionality. *Ontological correspondence:* the ontology of meta-level facilities should correspond to the ontology of the language they manipulate. Traditional/mainstream reflective architectures do not follow these precepts. In contrast, reflective APIs built around the concept of *mirrors* are characterized by adherence to these three principles. Consequently, mirror-based architectures have significant advantages with respect to distribution, deployment and general purpose metaprogramming.

Categories and Subject Descriptors

D.3.2 [Language Classifications]: Object-oriented languages.

General Terms

Design, Languages.

Keywords

Reflection, Metaprogramming, Mirrors, Java, Self, Smalltalk.

1. INTRODUCTION

Object-oriented languages traditionally support meta-level operations such as reflection by reifying program elements such as classes into objects that support reflective operations such as get Superclass or getMethods.

In a typical object oriented language with reflection, (e.g., Java, C#, Smalltalk, CLOS) one might query an instance for its class, as indicated in the pseudo-code below:

```
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:

```
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-level operations coexist side-by-side. The same class object that contains constructors and static attributes also responds to queries about its name, superclass, and members. 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:

```
class Object
// no re ective methods
...

class Class
// no re ective methods
...

interface Mirror
String name();
...

class Re ection
public static ObjectMirror re ect(Object o) ...

interface ObjectMirror extends Mirror
 ClassMirror getClass();
...

interface ClassMirror extends Mirror
 ClassMirror getSuperclass();
...
```

"We identify
three design principles for reflection
and metaprogramming facilities in
object-oriented programming languages. "

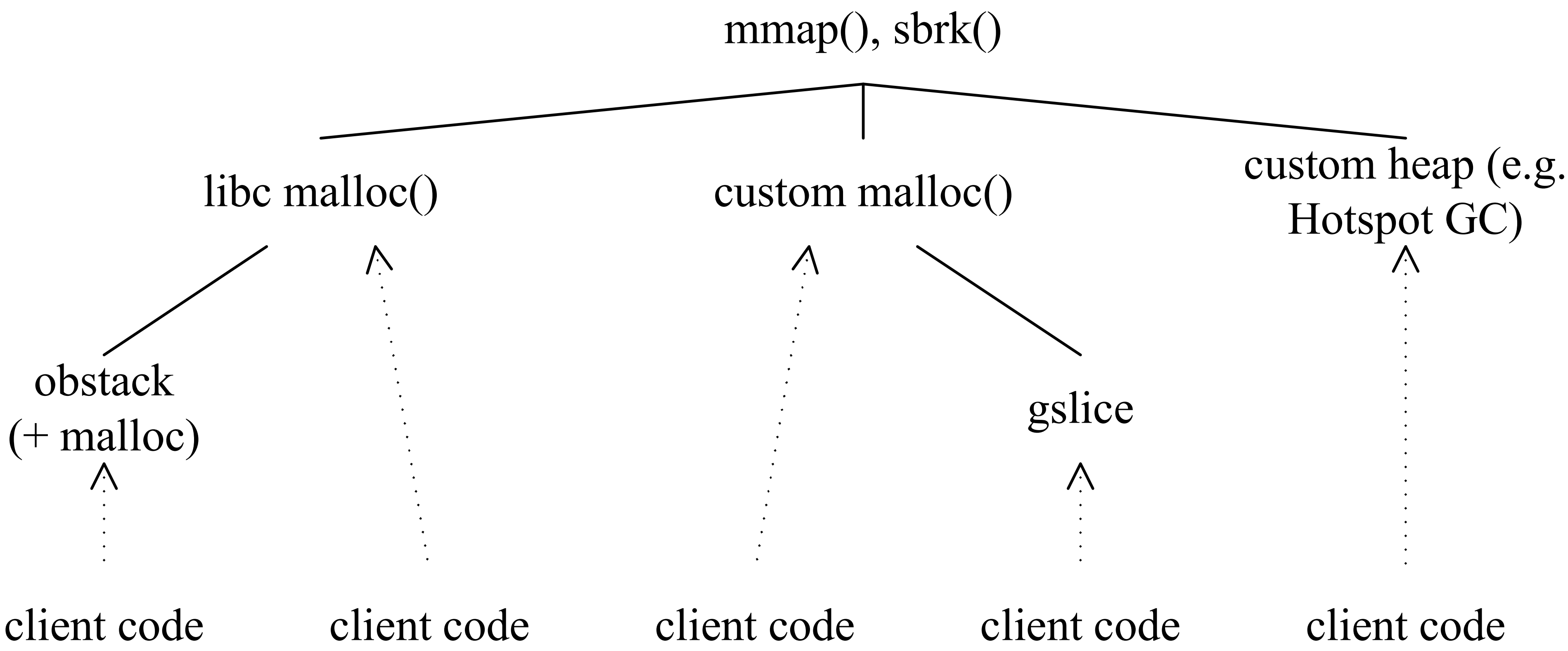
Bracha & Ungar
Mirrors: design principles for meta-level facilities of object-oriented programming languages
OOPSLA 2004

- Unix debugging adopted the same principles long ago

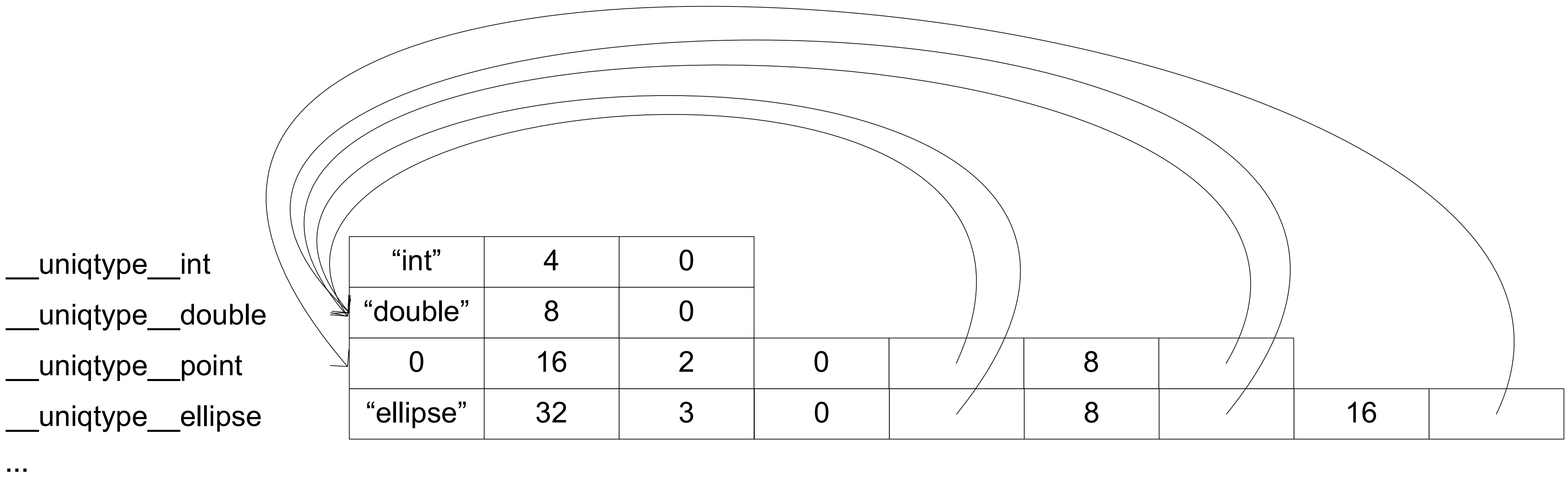
The liballocs meta-protocol

```
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.


mmap(), sbrk()
libc malloc()
custom malloc()
custom heap (e.g. Hotspot GC)
obstack (+ malloc)
gslice
client code
client code
client code
client code
client code

```
struct ellipse {
    double maj, min;
    struct point { double x, y; } ctr ;
};
```



+ lots not shown (named fields, functions, unions, . . .)

Compatibility with existing language implementations

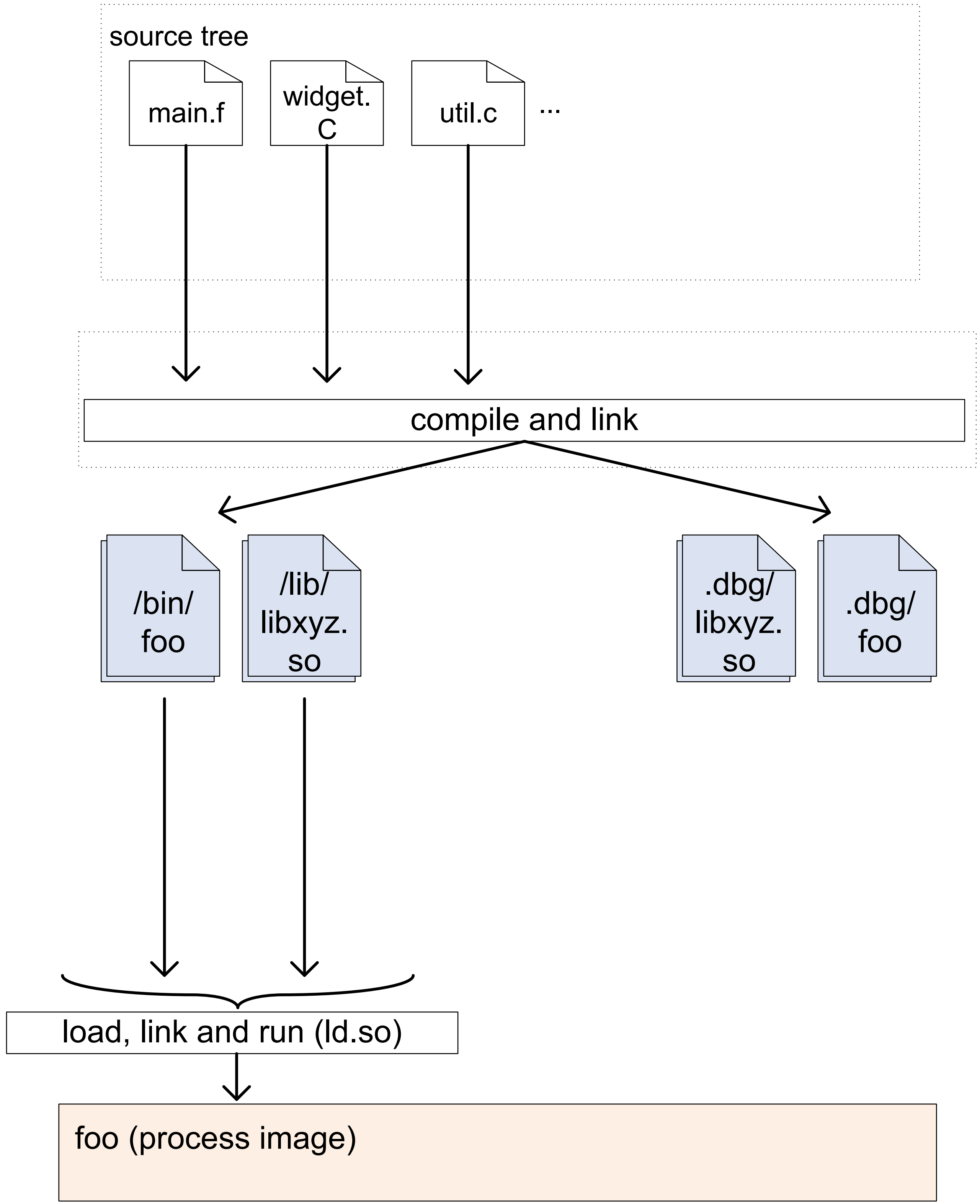
Two cases!

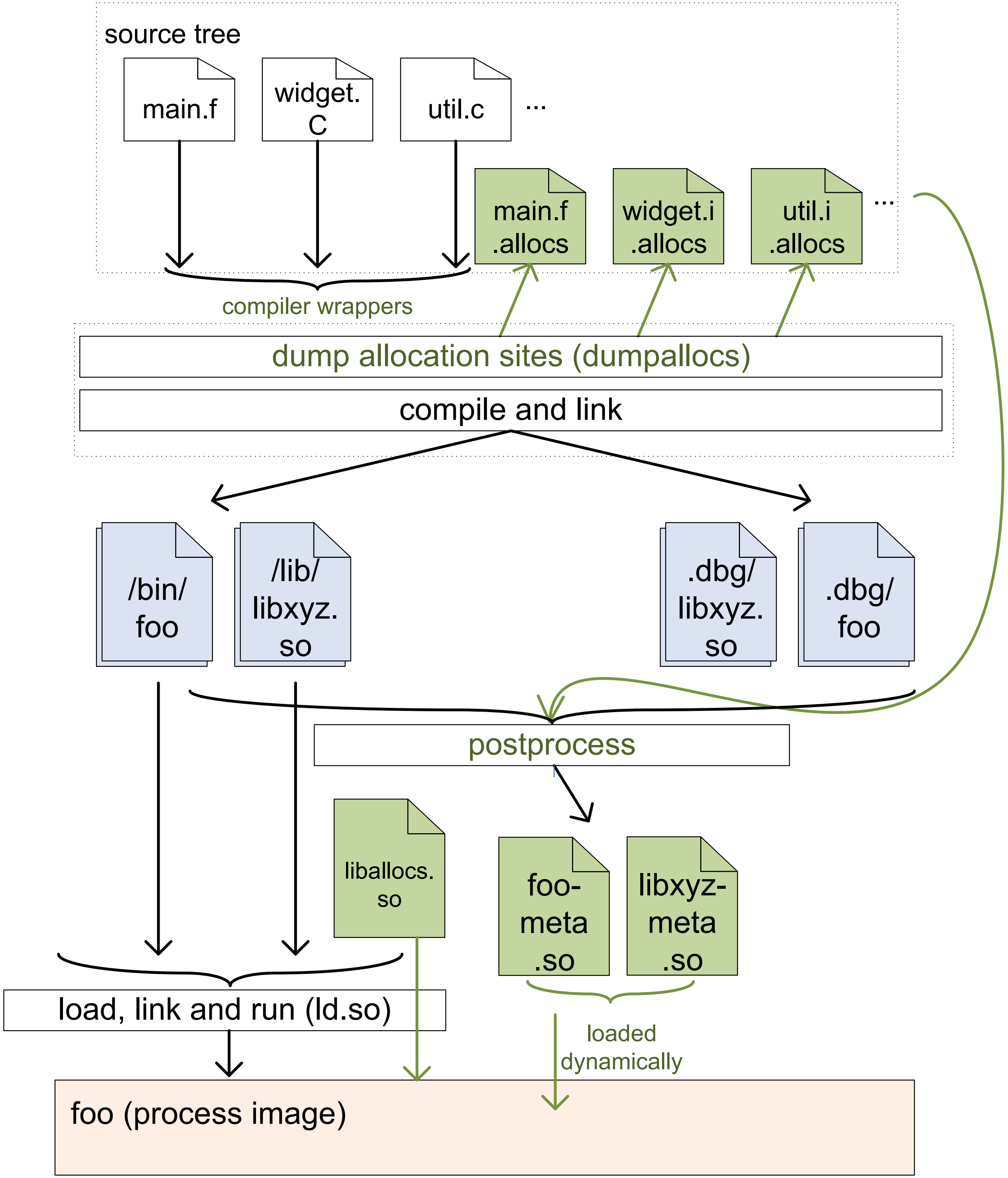
Unix-style compilation toolchains (C, C++, Fortran, . . .)

- *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.C
util.c
...
compile and link
/bin/foo
/lib/libxyz.so
.dbg/libxyz.so
.dbg/foo
load, link and run (ld.so)
foo (process image)



source tree
main.f
widget.C
util.c
...
main.f.allocs
widget.i.allocs
util.i.allocs
...
compiler wrappers
dump allocation sites (dumpallocs)
compile and link
/bin/foo
/lib/libxyz.so
.dbg/libxyz.so
.dbg/foo
postprocess
liballocs.so
foo-meta.so
libxyz-meta.so
load, link and run (ld.so)
loaded dynamically
foo (process image)

bench	normal/s	liballocs/s	liballocs %	no-load
bzip2	4.91	5.05	+2.9%	+1.6%
gcc	0.985	1.85	+88 %	– %
gobmk	14.2	14.6	+2.8%	+0.7%
h264ref	10.1	10.6	+5.0%	+5.0%
hmmer	2.09	2.27	+8.6%	+6.7%
lbm	2.10	2.12	+0.9%	(−0.5%)
mcf	2.36	2.35	(−0.4%)	(−1.7%)
milc	8.54	8.29	(−3.0%)	+0.4%
perlbench	3.57	4.39	+23 %	+1.6%
sjeng	3.22	3.24	+0.6%	(−0.7%)
sphinx3	1.54	1.66	+7.7%	(−1.3%)

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

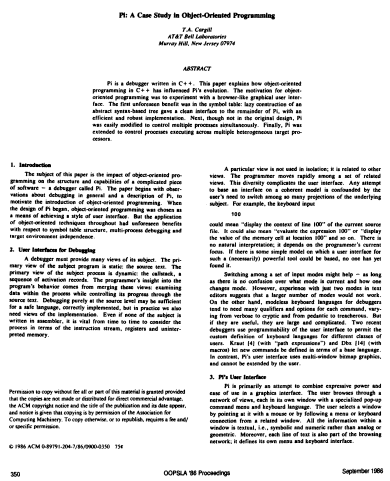
- whole-process reflection
- whole-process tools (debuggers, profilers, . . .)
- interop without FFIs (improving the node use-case)

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

- please get in touch

Thanks for listening. . . questions?

The real difference between Unix and VMs



Pi: A Case Study in Object-Oriented Programming

T.A. Cargill
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Pi is a debugger written in C++. This paper explains how object-oriented programming in C++ has influenced Pi's evolution. The motivation for object-oriented programming was to experiment with a browser-like graphical user interface. The first unforeseen benefit was in the symbol table: lazy construction of an abstract syntax-based tree gave a clean interface to the remainder of Pi, with an efficient and robust implementation. Next, though not in the original design, Pi was easily modified to control multiple processes simultaneously. Finally, Pi was extended to control processes executing across multiple heterogeneous target processors.

1. Introduction

The subject of this paper is the impact of object-oriented programming on the structure and capabilities of a complicated piece of software – a debugger called Pi. The paper begins with observations about debugging in general and a description of Pi, to motivate the introduction of object-oriented programming. When the design of Pi began, object-oriented programming was chosen as a means of achieving a style of user interface. But the application of object-oriented techniques throughout had unforeseen benefits with respect to symbol table structure, multi-process debugging and target environment independence.

2. User Interfaces for Debugging

A debugger must provide many views of its subject. The primary view of the subject program is static: the source text. The primary view of the subject process is dynamic: the callstack, a sequence of activation records. The programmer's insight into the program's behavior comes from merging these views: examining data within the process while controlling its progress through the source text. Debugging purely at the source level may be sufficient for a safe language, correctly implemented, but in practice we also need views of the implementation. Even if none of the subject is written in assembler, it is vital from time to time to consider the process in terms of the instruction stream, registers and uninterpreted memory.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1986 ACM 0-89791-204-7/86/0900-0350 75¢

A particular view is not used in isolation; it is related to other views. The programmer moves rapidly among a set of related views. This diversity complicates the user interface. Any attempt to base an interface on a coherent model is confounded by the user's need to switch among so many projections of the underlying subject. For example, the keyboard input

100

could mean "display the context of line 100" of the current source file. It could also mean "evaluate the expression 100" or "display the value of the memory cell at location 100" and so on. There is no natural interpretation; it depends on the programmer's current focus. If there is some simple model on which a user interface for such a (necessarily) powerful tool could be based, no one has yet found it.

Switching among a set of input modes might help – as long as there is no confusion over what mode is current and how one changes mode. However, experience with just two modes in text editors suggests that a larger number of modes would not work. On the other hand, modeless keyboard languages for debuggers tend to need many qualifiers and options for each command, varying from verbose to cryptic and from pedantic to treacherous. But if they are useful, they are large and complicated. Two recent debuggers use programmability of the user interface to permit the custom definition of keyboard languages for different classes of users. Kraut [4] (with "path expressions") and Dbx [14] (with macros) let new commands be defined in terms of a base language. In contrast, Pi's user interface uses multi-window bitmap graphics, and cannot be extended by the user.

3. Pi's User Interface

Pi is primarily an attempt to combine expressive power and ease of use in a graphics interface. The user browses through a network of views, each in its own window with a specialized pop-up command menu and keyboard language. The user selects a window by pointing at it with a mouse or by following a menu or keyboard connection from a related window. All the information within a window is textual, i.e., symbolic and numeric rather than analog or geometric. Moreover, each line of text is also part of the browsing network; it defines its own menu and keyboard interface.

350 OOPSLA '86 Proceedings September 1986

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
 <b>:TAG_compile_unit              <7ae>:TAG_pointer_type
      AT_language  : 1 (ANSI C)           AT_byte_size: 8
      AT_name      : hello.c              AT_type     : <0x2af>
      AT_low_pc    : 0x4004f4     <76c>:TAG_subprogram
      AT_high_pc   : 0x400514             AT_name     : main
 <c5>: TAG_base_type                      AT_type     : <0xc5>
        AT_byte_size : 4                  AT_low_pc   : 0x4004f4
        AT_encoding  : 5 (signed)         AT_high_pc  : 0x400514
        AT_name      : int         <791>: TAG_formal_parameter
 <2af>:TAG_pointer_type                    AT_name     : argc
        AT_byte_size: 8                    AT_type     : <0xc5>
        AT_type     : <0x2b5>              AT_location : fbreg - 20
 <2b5>:TAG_base_type               <79f>: TAG_formal_parameter
        AT_byte_size: 1                    AT_name     : argv
        AT_encoding : 6 (char)             AT_type     : <0x7ae>
        AT_name     : char                 AT_location : fbreg - 32
```

Retrofitting dynamic compilation

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

What about dynamic compilers?

```
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
```