Virtual machines should be invisible (and might be augmented)

Stephen Kell

stephen.kell@cs.ox.ac.uk

some joint work with

Conrad Irwin (University of Cambridge)

Spot the virtual machine (1)

Spot the virtual machine (2)

Spot the virtual machine (3)

(Hint: they're all invisible)

Hey, you got your VM in my Programming Experience™!

VMs don't support programmers; they *impose on* them:

- limited language selection
- "foreign" code must conform to FFI
- debug with *per-VM* tools (jdb? pdb?)
- developing *across* VM boundaries? forget it!

Wanted:

- an end to FFI coding in the common case (assuming...)
- tools that work *across* VM boundaries

Focus on dynamic languages (→ Python for now)...

Warning and apology

How we're going to do it

Conventional VMs: "cooperate or die!"

- you will conform
- you will use my tools

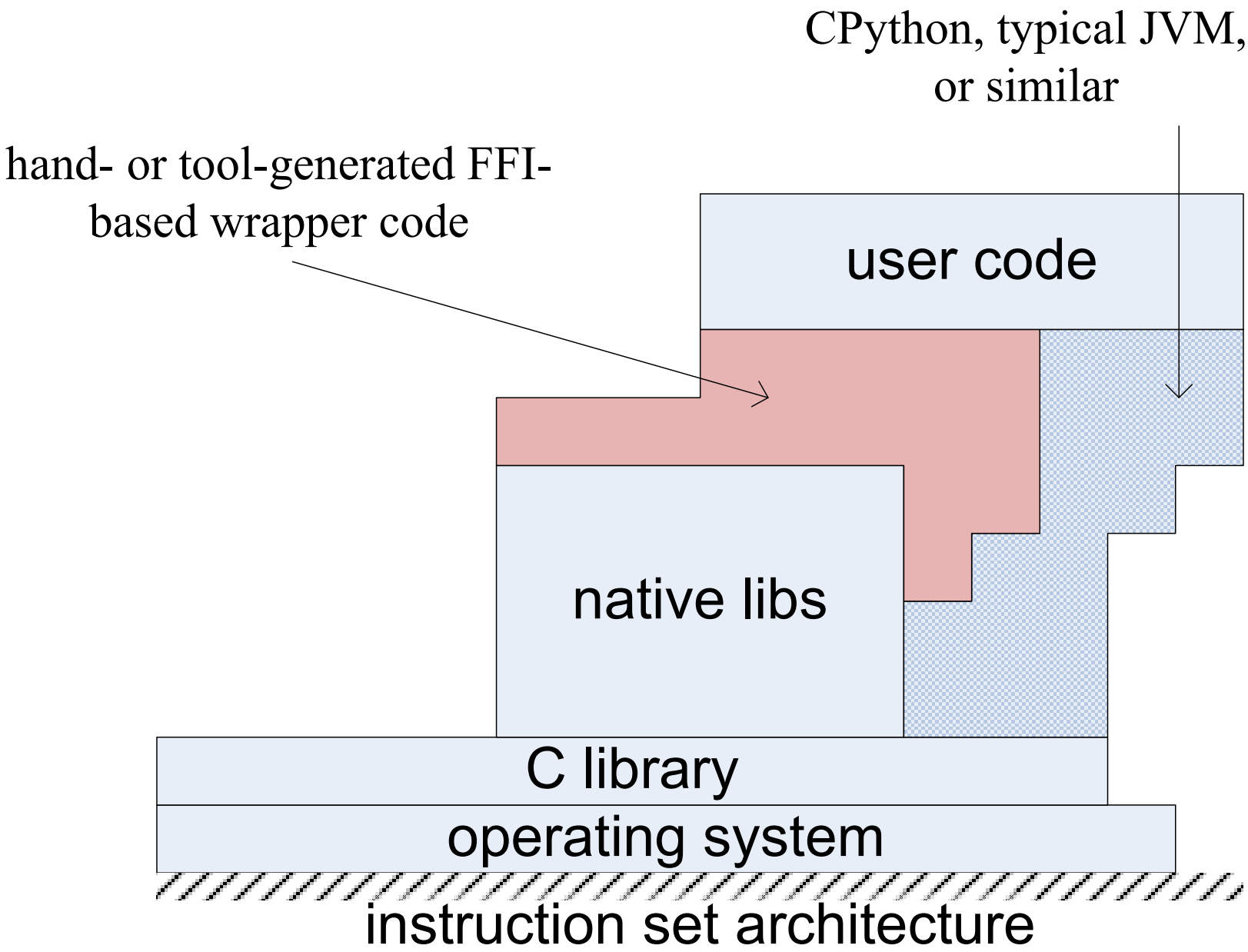
"Less obtrusive" VMs:

- "Describe yourself, alien!"
- … and I'll describe myself (to *whole-process* tools)

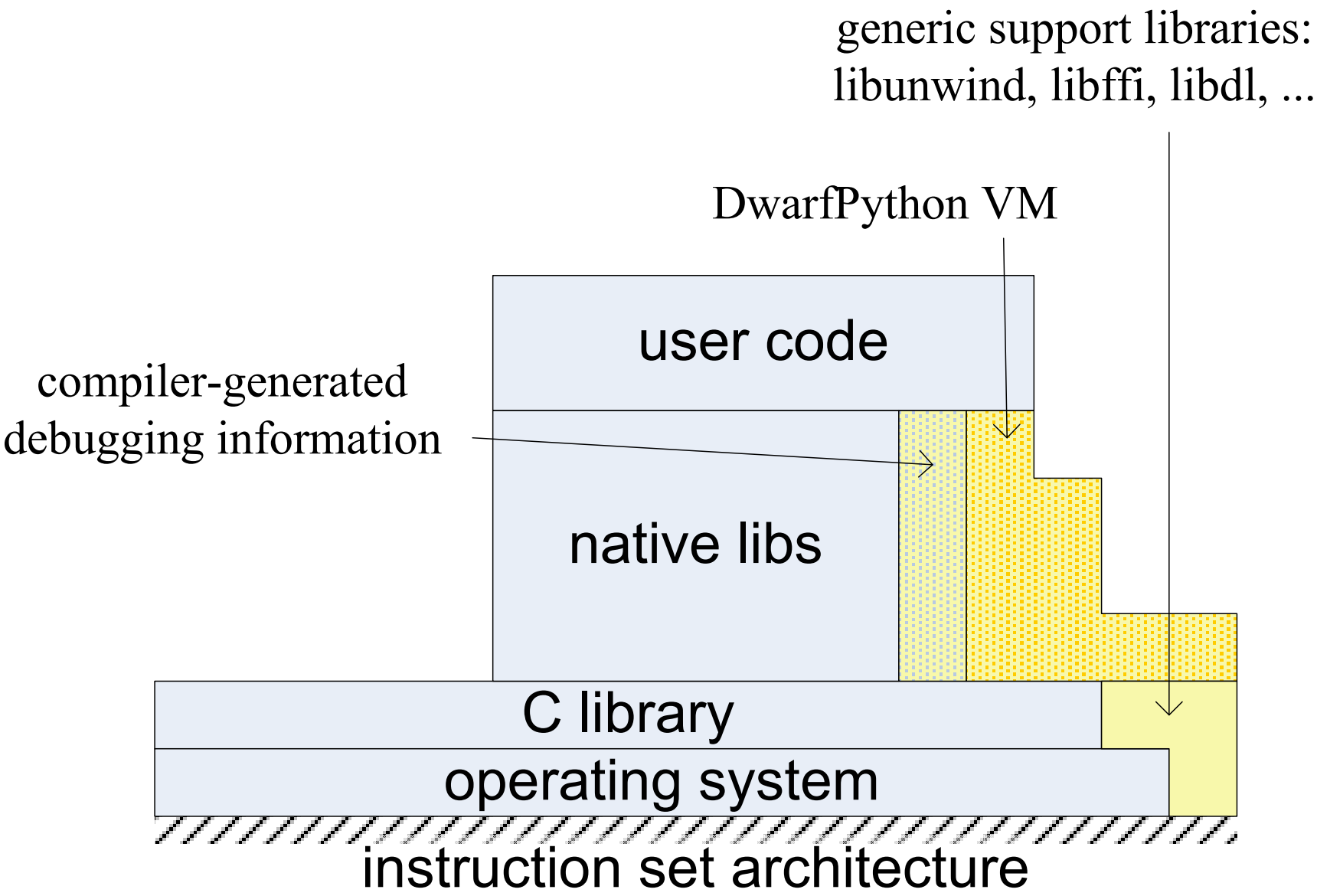
In particular:

- extend *underlying* infrastructure: libdl, malloc, …
- … and a *shared descriptive metamodel*—DWARF!
- never (re)-invent opaque VM structures / protocols!

Implementation tetris (1)



Implementation tetris (2)



DwarfPython: an unobtrusive Python VM

DwarfPython is an ongoing implementation of Python which

- can import native libraries as-is
- can share objects directly with native code
- supports debugging with native tools

Key components of interest:

- unified notion of function as *entry point(s)*
- extended libdl sees *all* code; entry point generator
- extensible objects (using DWARF + extended malloc)
- interpreter-created objects described by DWARF info

No claim to fully-implementedness (yet)…

What is DWARF anyway?

```
$ 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
```

Functions as black boxes

Functions are *loaded*, *named* objects:

- extend libdl for dynamic code: dlcreate(), dlbind(), …
- no functions “foreign” (our impl.: always use libffi)

```
def fac:
    if n == 0: return 1
    else: return n * fac(n-1)
```

```
0x2aaaaf640000 <fac>:
 00:  push %rbp
;  -- snip
 23:  callq *%rdx
;  -- snip
 2a:  retq
```

```
<b>: TAG_compile_unit
<10>  AT_language: 0x8001(Pythor
<11>  AT_name    : dwarfpy REPL
<f6>:TAG_subprogram
<76e> AT_name    :  fac
<779> AT_low_pc  : 0x2aaaaf6400
<791>:TAG_formal_parameter
<792>  AT_name    : n
<79c>  AT_location: fbreg - 20
```

What have we achieved so far?

Make VMs responsible for generating entry points; then

- in-VM code is not special (can call, dlsym, . . .)
- host VM and impl. language are “hidden” details

What’s left?

- exchanging data, sharing data
- making debugging tools work
- many subtleties (ask me if I don’t cover yours)

Accessing and sharing objects

Objects don't "belong" to any VM. They are just memory…

- … *described* by DWARF.

Jobs for VMs and language implementations:

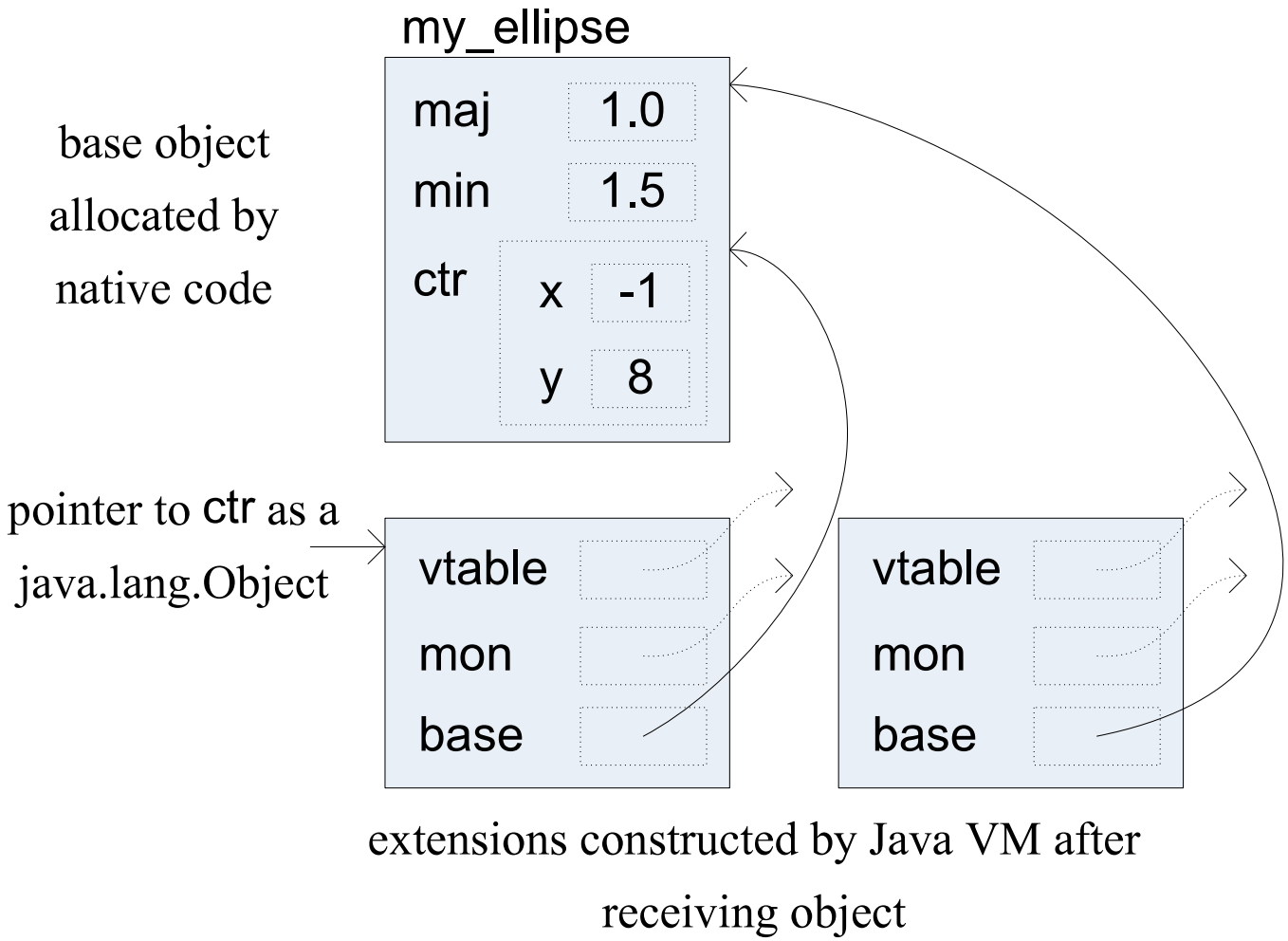
- Map each language's data types to DWARF (as usual)
- Make sense of arbitrary objects, dynamically.
  - Python: mostly easy enough (like a debugger)
  - Java: need to java.lang.Objectify, dynamically

Assumption: can map any pointer to a DWARF description.

- use some fast malloc instrumentation

Java-ifying an object created by native code

- object extension
- . . . dynamically
- non-contiguous
- tree-structured
- “fast” entry pts can skip this



extensions constructed by Java VM after receiving object

Wrapping up the object model

Summary: invisible VMs take on new responsibilities:

- describe objects they create; accommodate others
- register functions with libdl ($\rightarrow$ generate entry points!)

Lots of things I haven't covered; ask me about

- garbage collection
- dispatch structures (vtables, . . .)
- reflection (but you can guess)
- extensions to DWARF
- memory infrastructure
- abstraction gaps between languages

Doing without FFI code: a very simple C API

```
static PyObject* Buf_new(
 PyTypeObject* type, PyObject*
 args, PyObject* kwds) {
  BufferWrap* self;
  self = (BufferWrap*)type->
   tp_alloc(type, 0);
  if (self != NULL) {
   self->b = new_buffer();
   if (self->b == NULL) {
    Py_DECREF(self);
    return NULL;
 } }  return (PyObject*)self; }
```

– CPython wrapper

– allocate type object (1)

– call underlying func (2)

– adjust refcount (3)

VM can do all this *dynamically*!

- … given ABI *description*

Can be interpreted, or used as input to dynamic compilation

What about debugging?

```
(gdb) bt
#0  0x0000003b7f60e4d0 in __read_nocancel () from /lib64/libp
#1  0x00002aaaace3f7c5 in ?? ()
#2  0x00002aaaaaa3b7b3 in ?? ()
#3  0x0000000000443064 in main (argc=1, argv=0x7fffffffd828)
```

We need to fill in the question marks. Easy!

- handily, everything is described using DWARF info
- . . . with a few extensions
- . . . just tell the debugger how to find it!
- anecdote / contrast: LLVM JIT + gdb protocol

Why it works: the dynamism–debugging equivalence

debugging-speak	runtime-speak
backtrace	stack unwinding
state inspection	reflection
memory leak detection	garbage collection
altered execution	eval function
edit-and-continue	dynamic software update
breakpoint	dynamic weaving
bounds checking	(spatial) memory safety

A debuggable runtime is a dynamic runtime.

Dynamic reasoning is our fallback.

Even native code should be debuggable!

What about performance? What about correctness?

Achievable performance is an open question. However,

- our heap instrumentation is fast
- intraprocedural optimization unaffected

We can now do *whole-program dynamic optimization*!

- libdl is notified of optimized code
- VM supplies *assumptions* when generating code…

Correctly enforcing invariants is a whole-program concern!

- "guarantees" become "assume–guarantee" pairs
- e.g. "if caller guarantees P, I can guarantee Q"
- libdl is a good place to manage these too

Status

Lots of implementation is not done yet! Some is, though.

- libpmirror, DWARF foundations: functional (but slow)
- memory helpers (libmemtie, libmemtable) near-done
- extended libdl: proof of concept
- dwarfpython: can *almost* do fac!
- parathon (predecessor), usable subset of Python

Lots to do, but…

…I think we can make virtual machines less obtrusive!

The rest of this talk

There's something about VMs

VM *implementations* are mostly concerned with

- efficiently realising "virtual" abstractions, concretely
- $\rightarrow$ GC, bytecode, dynamic optimization, ...

But also, VMs are concerned with *protecting* abstractions:

- ... e.g. type safety properties
- by "on-line reasoning" (dynamic checks)

Crazy idea: also use VM infrastructure for *off-line reasoning*!

Off-line a.k.a. “static” reasoning? But ~~how?~~ why?

Benefits on the agenda:

- comprehensible error messages
- tunable precision (abstract only when necessary)
- specify in user language
- support many (incl. dynamic) languages
- programmer guidance becomes easier?
- easier to make it go fast?
- compositional w.r.t. kinds of check?

Contrast: traditional type-checking…

Virtual versus augmented

The “how” questions

Q1. What can we specify and check. . .

- . . . *on*-line, for now. . .
- . . . on current VM infrastructure?
- How to extend this?

Q2. How can we make it work also for off-line reasoning. . .

- . . . i.e. “static” reasoning?
- How well? Will it terminate? Will it be sound?

First stop: Q1, i.e. understanding what we can specify.

Protecting abstractions at run time

On-line reasoning requires *specification*. . .

- "only interpret memory under its allocated *data type*!"

. . . and, at run time, maintaining *descriptive* info:

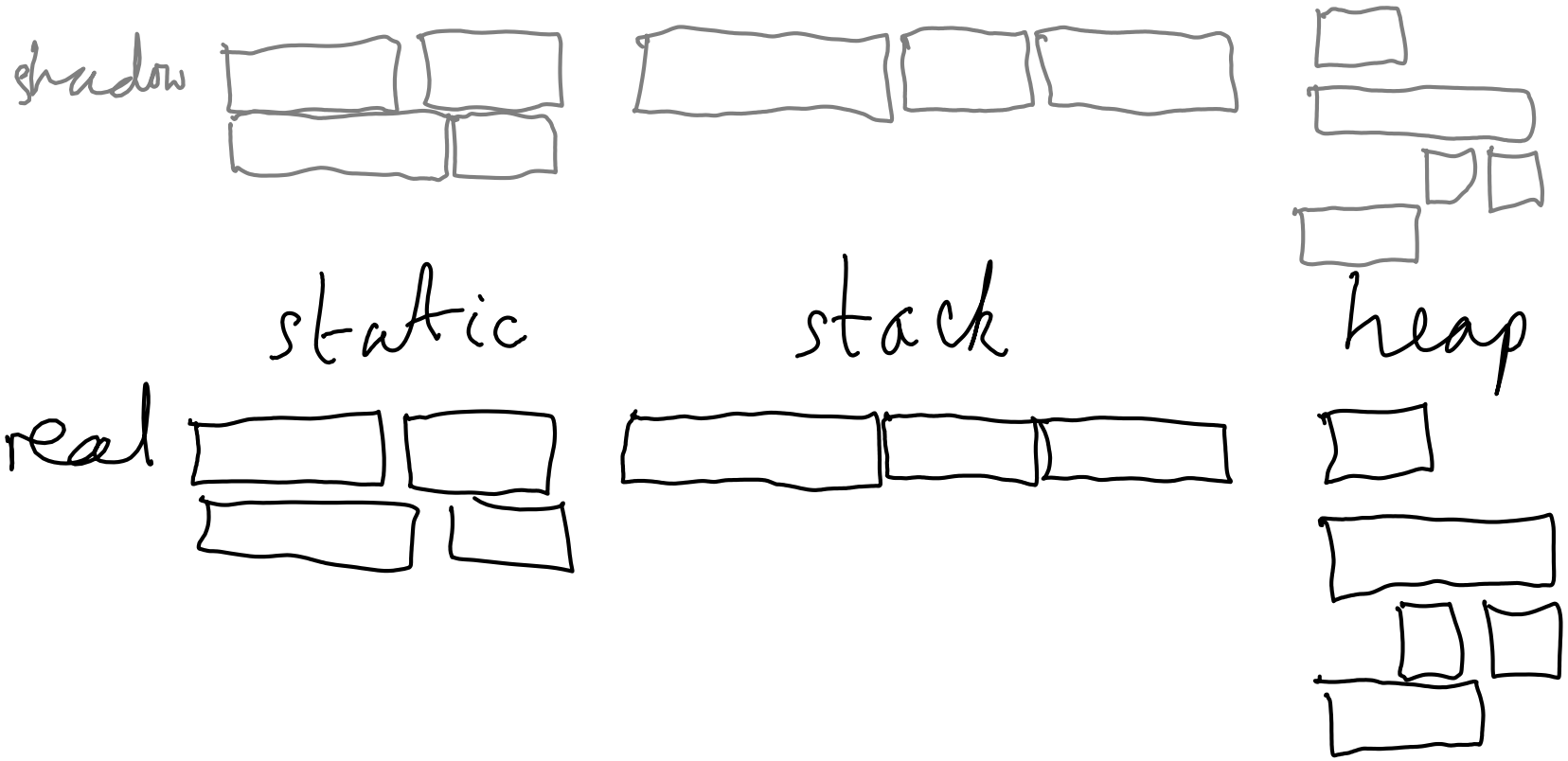
- "this memory was allocated as this data type!"

Run-time type info is *one* kind of description; others:

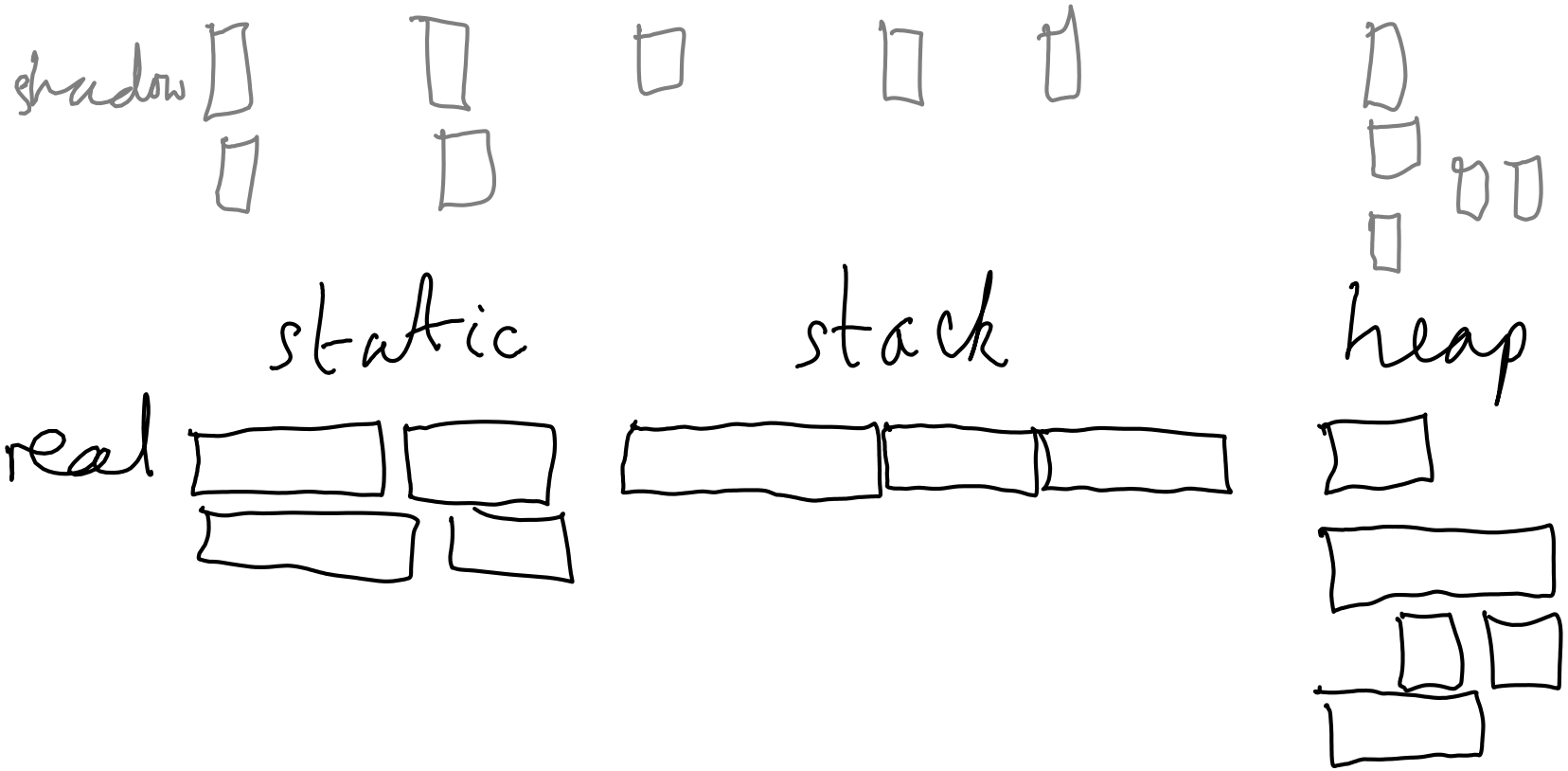
- determinism
- performance (e.g. timestamps, cache sim, . . .)
- provenance/influence, . . .
- note: not just for correctness!

Shadow value tools (Valgrind et al) track these properties. . .

Shadow value analysis: memcheck



Shadow value analysis: ~~memecheck~~ type check



VMs augmented with shadowy metadata. . .

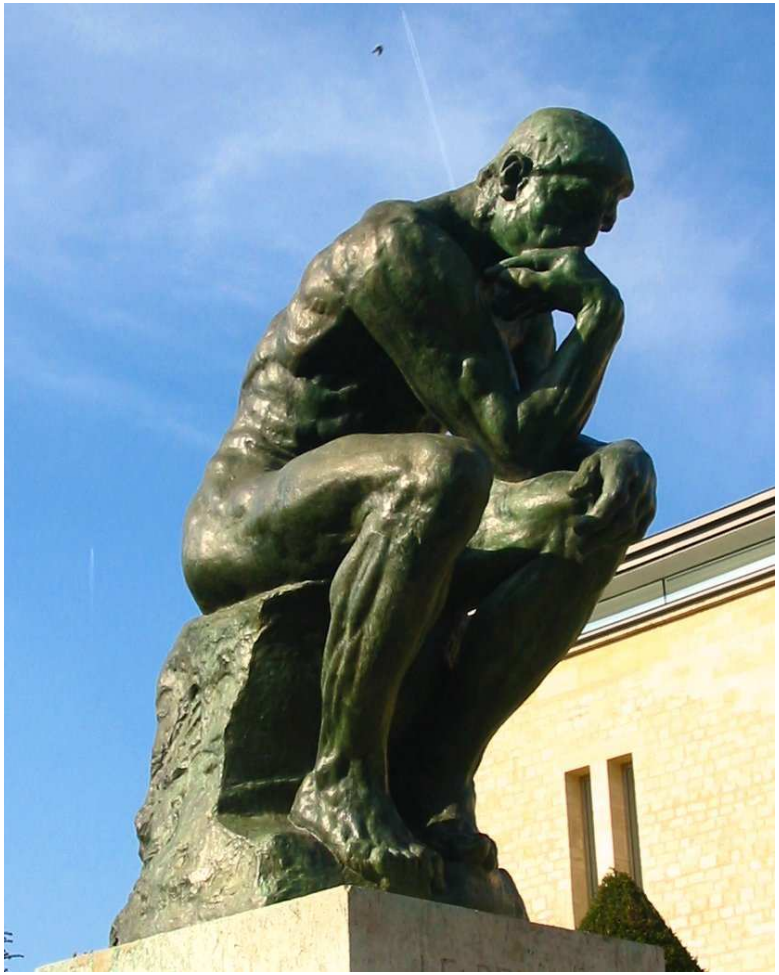
So far this is just testing + instrumentation; nice properties:

- (trivially) no false positives
- errors are concrete $\longrightarrow$ comprehensible
- shadowing is good fit for VM infrastructure
- specification domain is programmer-friendly
- composition: multiple shadows easily coexist
  - vertical composition too: typestate (others?)

Bad things:

- memory overhead
- time overhead
- single runs only! we can fix this. . .

Static versus dynamic analysis



Symbolic execution: the Noddy story (1)

Symbolic execution is a forward (sp) analysis:

- represent program state in input language of SMT solver
- symbolic variables represent arbitrary input
- systematic branching exploration of state space

Apply to bug-finding ("test case generation")

- log feasible failures; solver can generate test input
- usually runs forever…

Is this static or dynamic?

- both! neither!

Symbolic execution: the Noddy story (2)

```
int main(int argc, char **argv) // ... where argv[1] is symbolic
{
  assume(argc > 1 && strlen(argv[1]) == 2); // narrow to two-char cases
  int temp1 = atoi(argv[1]);
}
```

At the end of this function, temp1 has a symbolic value:

```
(Add w32
   (Mul w32 10 (SExt w32 (Read w8 0 argv_1)))
   (SExt w32 (Read w8 1 argv_1)))
```

Execution is

- *deferred*, w.r.t. program values
- *exploratory*, w.r.t. program paths

What to do

Symbolic execution tools only find errors that are

- generic (null pointer, out-of-bounds, div-by-zero,...), or
- programmer-supplied: assert() over program variables

Shadowing lets us expand our specification language!

- defined(), tainted(), owner(), aliases(), fperror(),...
- your idea here! (+ some unusual related work)
- tentative claim: these are easy for programmers to grok.

Use symbolic execution to “accelerate” dynamic analyses!

Starting small: a type-checking-like analysis

Idea: build an analysis competitive with type-checking?

- retaining our benefits:
  - can tune avoid false positives…
  - … or be more overapproximating
  - no obligation to produce a proof
  - … not-provably-true checks are “left in”
- … sufficiently scalably?

The last part is going to take some *abstraction*.

Reconstructing type checking

A naive symbolic execution might not reach line 6.

```
0  void *expensive(void *arg); // no type signature!
0  if (runExpensive) { outputStr = expensive(); }
0
0  else { outputStr = "(not␣done)"; }
0
0  assert(is_a(outputStr, "char"));
0  printf ("Status:␣%s\n", outputStr);
```

Want to avoid expensive(); how? By

- signatures (what type checkers do)

Reconstructing type checking

A naive symbolic execution might not reach line 6.

```
0  void *expensive(void *arg); // no type sig, but now have summary
0  if (runExpensive) { outputStr = expensive();
0                      shadow(outputStr) = "char"; // from summary
0  else   { outputStr = "(not␣done)";
0           shadow(outputStr) = "char"; }
0  assert(shadow(outputStr) == "char");
0   printf ("Status:␣%s\n", outputStr);
```

Want to avoid expensive(); how? By

- signatures (what type checkers have)
- summaries (their generalisation to symbolic execution)

Reconstructing type checking

A naive symbolic execution might not reach line 6.

```
0
0   if  (runExpensive) {
0                              shadow(outputStr) = ”char”; //  from summary
0   else    {
0                 shadow(outputStr) = ”char”; }
0   assert(shadow(outputStr) == ”char”);
0   //   printf ()  sliced  away
```

Want to avoid expensive(); how? By

- signatures (what type checkers have)
- summaries (their generalisation to symbolic execution)

To check *just* the assertions:

- slice on the assertion condition!

Exercise: make it check null-termination also. . .

Summary so far (1)

What's good about all this?

- extensible specifications, rooted in host language
- can ensure true positives ($\rightarrow$ good error messages)
- compositional: many shadows can co-exist

Conjectures:

- shadow values are friendlier than complex type systems
- summaries and slice-style abstractions are friendly too
  - consider a programmer debugging failure of the tool…

Summary so far (2)

Challenges:

- making it terminate
- DSL for specifying new shadow domains?
- integrating with dynamic compilation infrastructure

Scalability is dependent on *dependency structure*

- i.e. how complex the relationship between…
- …data-dependencies of shadow values…
- …and program variables
- … hence determining how much can be sliced away

Thanks for listening. Any questions?