How to make a virtual machine less virtual

Or: an "integrated" approach to dynamic language implementation

Stephen Kell

stephen.kell@comlab.ox.ac.uk

UNIVERSITY OF OXFORD

Programming languages…

Programming languages are great, but…

- requirements are diverse ("none is perfect")
- even *within one program*!

However,

- incorporating foreign code is costly
  - (think JNI, Python C API, Swig, …)
- *per-language* debugging tools are a poor solution
  - programmer burden; lack whole-program view
- performance suffers
  - reimplementation $\nrightarrow$ re-optimisation

One-slide summary of this talk

For the rest of this talk, I'll

- describe an approach for tackling these problems
- by changing how we implement higher-level languages
- focusing on the case of *dynamic languages*
- based on aggressive re-use of existing infrastructure
- … esp. of *debugging*
- "the process is the VM"
- zoom in on the memory management bit
- relate it to my mainline work

This work is ongoing, unfinished, background, hangover, …

Unifying infrastructures help

"Isn't this already solved?"

- JVM, CLR et al. unify many languages…
- "unify"ing FFI and debugging issues

But we could do better:

- what about *native* code? C, C++, …
- not all languages available on all VMs
- … FFI coding is still a big issue

What's the "most unifying" infrastructure?

What's in a virtual machine?

A virtual machine comprises...

- support for language implementors
  - GCing allocator; interpreter/JIT of some kind
  - object model: "typed", flat...
  - ... on heap only
- support for end programmers, coding
  - *core* runtime library (e.g. reflection, loader, ...)
  - "native interface" / FFI
- support for end programmers, debugging / "reasoning"
  - interfaces for debuggers, ...
- support for users / admins (security, res. man't, ...)

What's in ~~a virtual machine?~~ an OS process + minimal libc?

~~A~~ The "null" virtual machine comprises. . .

- support for language implementors
  - ~~GCing~~ allocator; ~~interpreter/JIT of some kind~~
  - object model: ~~"typed", flat~~ opaque. . .
  - . . . on heap ~~only~~ or stack or bss/rodata
- support for end programmers, coding
  - *core* runtime library (e.g. ~~reflection~~, loader, . . .)
  - "~~native~~ interface" / ~~F~~FI
- support for end programmers, debugging / "reasoning"
  - interfaces for debuggers, . . . at *whole process* scale
- support for users / admins (security, res. man't, . . .)

Astonishing claim

For most omissions, we can plug in libraries:

- JIT/interpreter…
- choose a GC (Boehm; can do better?)

Wha about reflection?

- … more generally, “dynamic” features

Debugging infrastructure supports all kinds of dynamism:

- name resolution, dynamic dispatch, …
- object schema updates (with some work)

… on *compiled* code, in *any* (compiled) language!

Well, almost…

Building "null VM" Python means plugging a few holes:

- … that are *already* problems for debuggers!
- that fit neatly into runtime and/or debugger facilities

I'm going to focus on a "hole".

- For the rest, ask me (or trust me…)

Some equivalences

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

For each pair, implement using the same infrastructure...

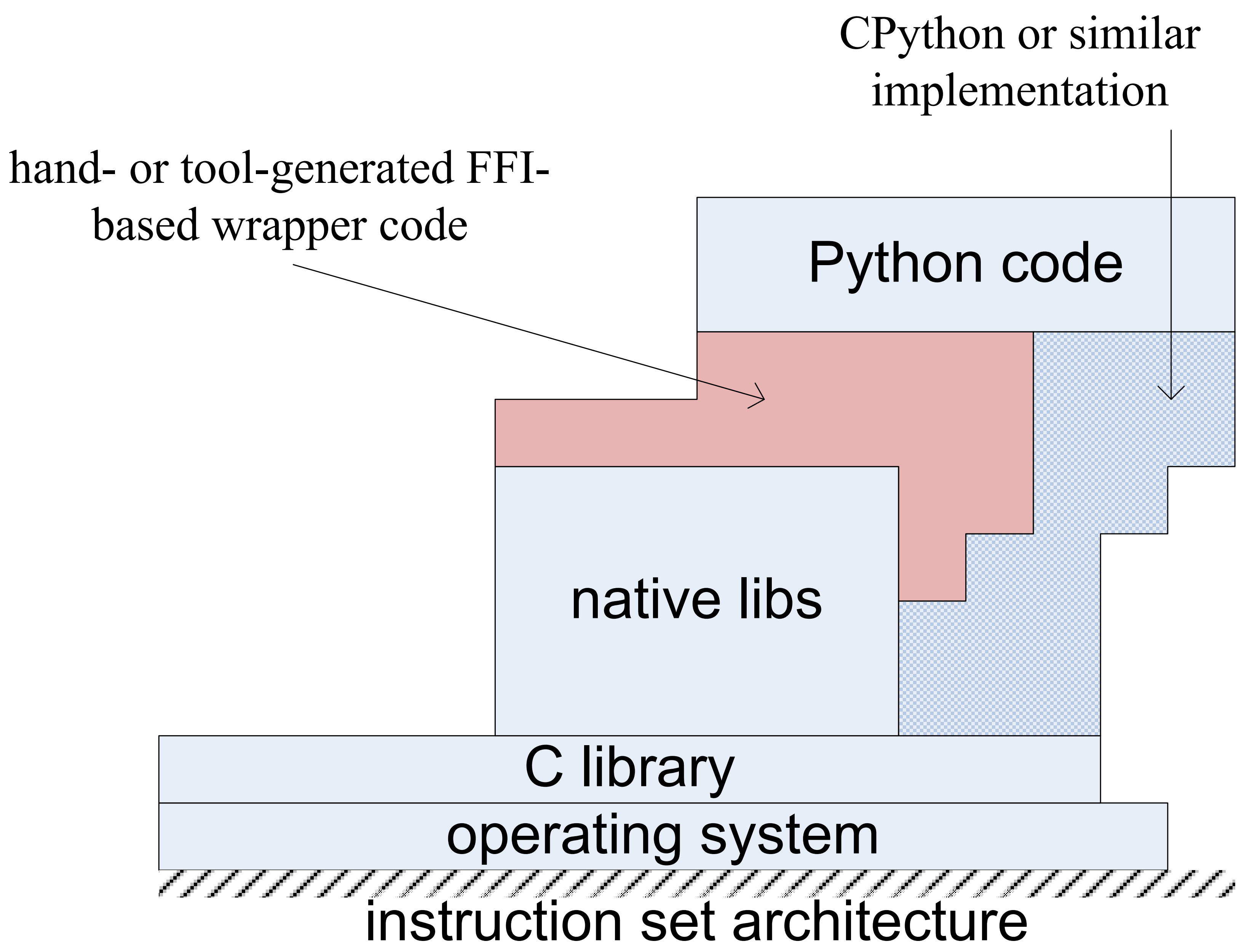
DwarfPython in one slide

DwarfPython is an implementation of Python which

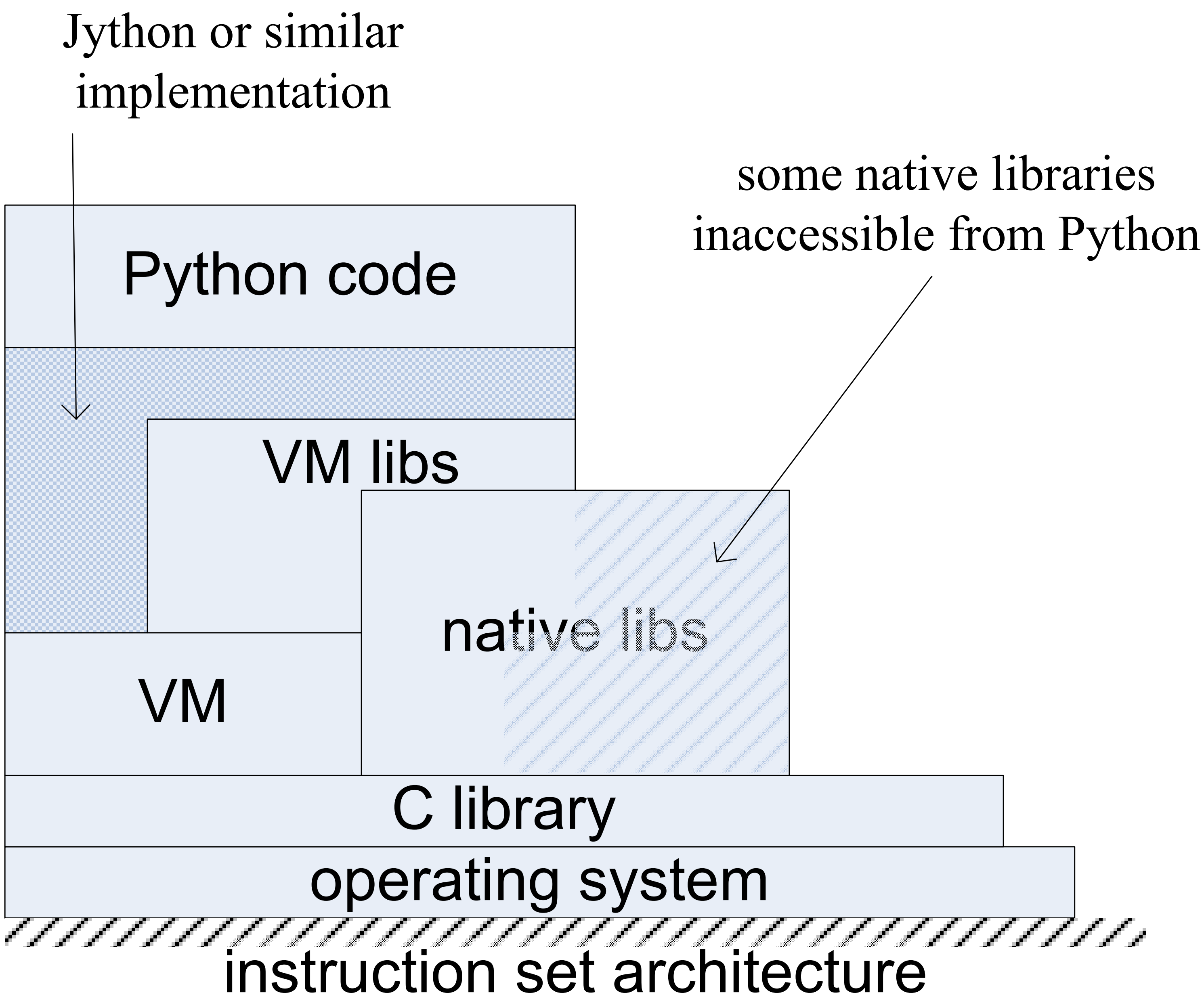
- uses DWARF debug info to understand native code…
  - … and itself!
- unifies Python object model with native (general) model
  - this is key!
- small, uniform changes allow `gdb`, `valgrind`, …
  - as a consequence of above two points
- deals with other subtleties…
  - I count 19 “somewhat interesting” design points

Not (yet): parallel / high-perf., Python *libraries*, …

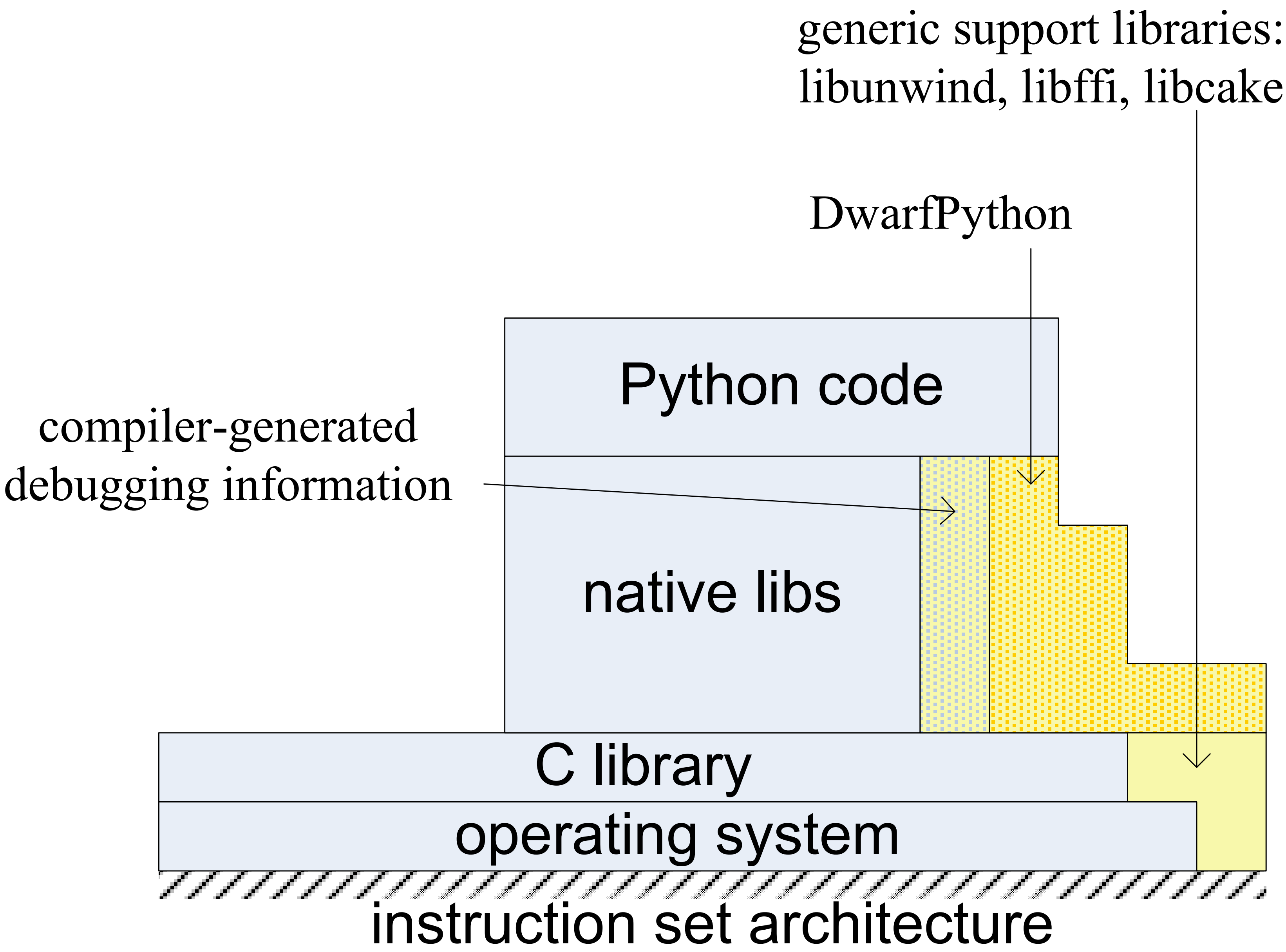
Implementation tetris (1)



Implementation tetris (2)



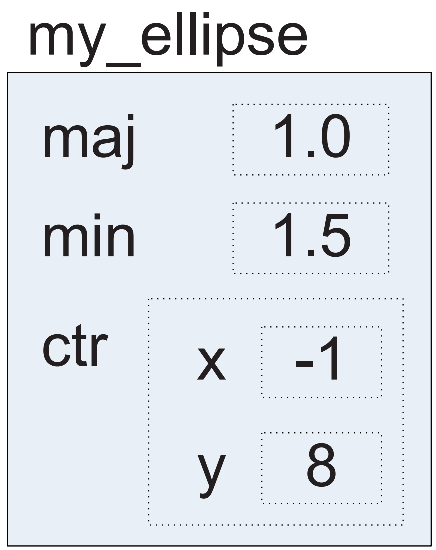
Implementation tetris (3)



Objects are not really opaque…

```
>>> import ellipse # dlopen()s libellipse.so
>>> my_ellipse = native_new_ellipse()
>>> print my_ellipse
```

Invariant 1: all objects have DWARF layout descriptions…




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

2d: DW_TAG_structure_type
  DW_AT_name : point
39: DW_TAG_member
  DW_AT_name : x
  DW_AT_type : <0x52>
  DW_AT_location: (DW_OP_plus_uconst: 0
45: DW_TAG_member
  DW_AT_name : y
  DW_AT_type : <0x52>
  DW_AT_location: (DW_OP_plus_uconst: 8
52: DW_TAG_base_type
  DW_AT_byte_size : 8
  DW_AT_encoding : 4 (float)
  DW_AT_name : double
59: DW_TAG_structure_type

Calling functions

```
>>> import c # libc.so already loaded
>>> def bye():  print "Goodbye, world!"
...
>>> atexit(bye)
```

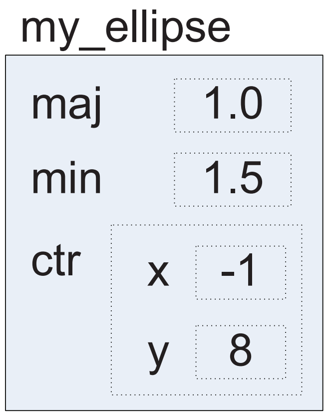
Invariant 2: *all* functions have ≥ 1 “native” entry point

- for Python code these are generated at run time

DwarfPython uses `libffi` to implement *all* calls

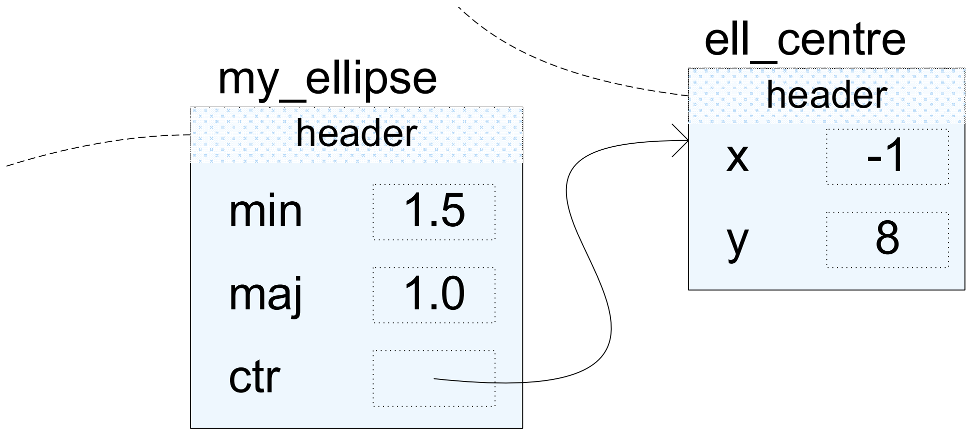
Object models

Dynamic dispatch means finding object metadata. Problem!




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

Native objects are trees; no descriptive headers, whereas. . .



VM-style objects: “no interior pointers” + custom headers

Wanted: fast metadata lookup

How can we locate an object's DWARF info

- … without object headers?
- … given possibly an *interior* pointer?

Solution:

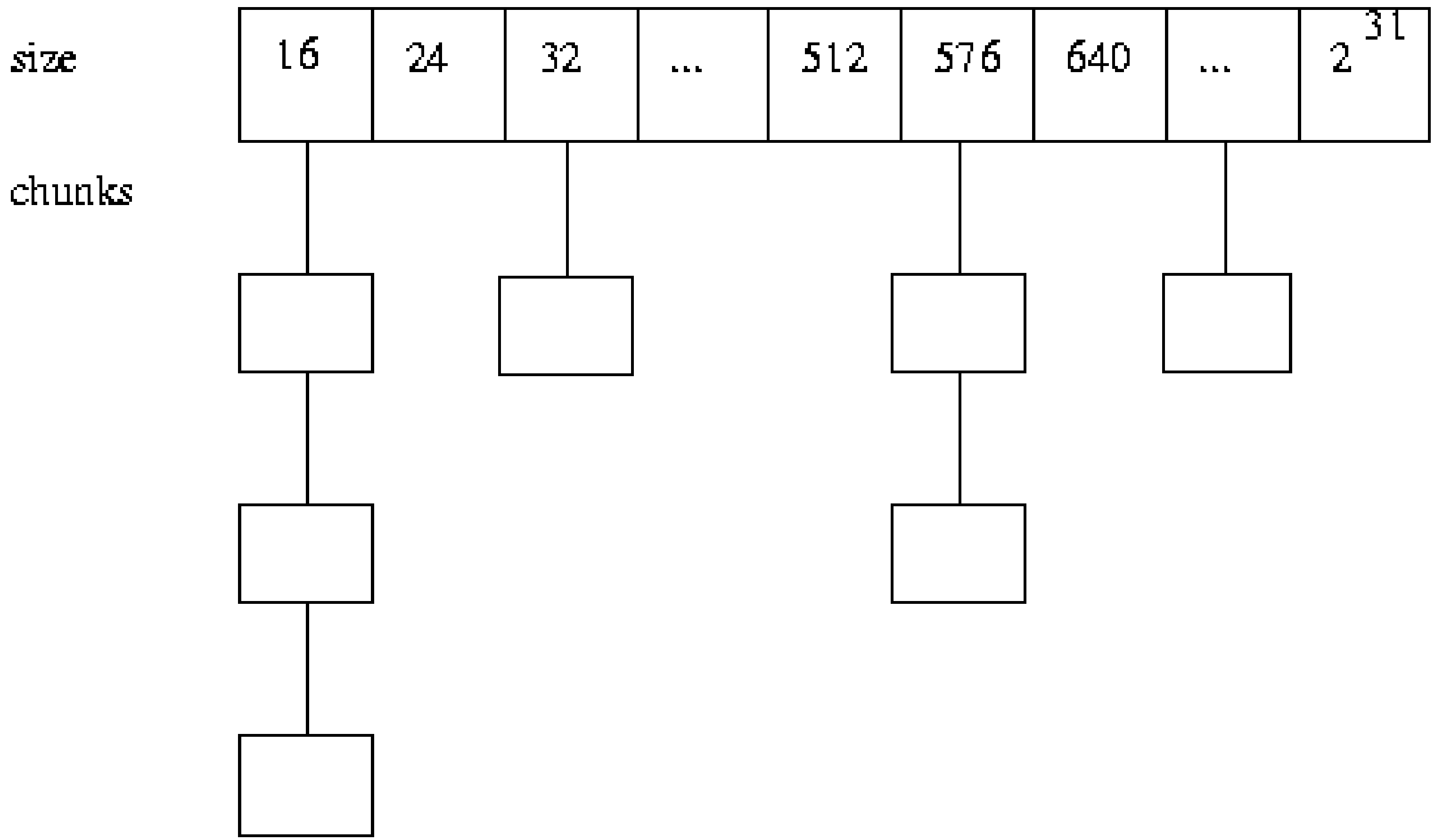
- is object on stack, heap or bss/rodata? ask memory map
- if static or stack, just use debug info (+ stack walker)

In the heap (difficult) case:

- we'll need some malloc() hooks…
- … and a *memtable*.
    - read: efficient *address-keyed* associative structure

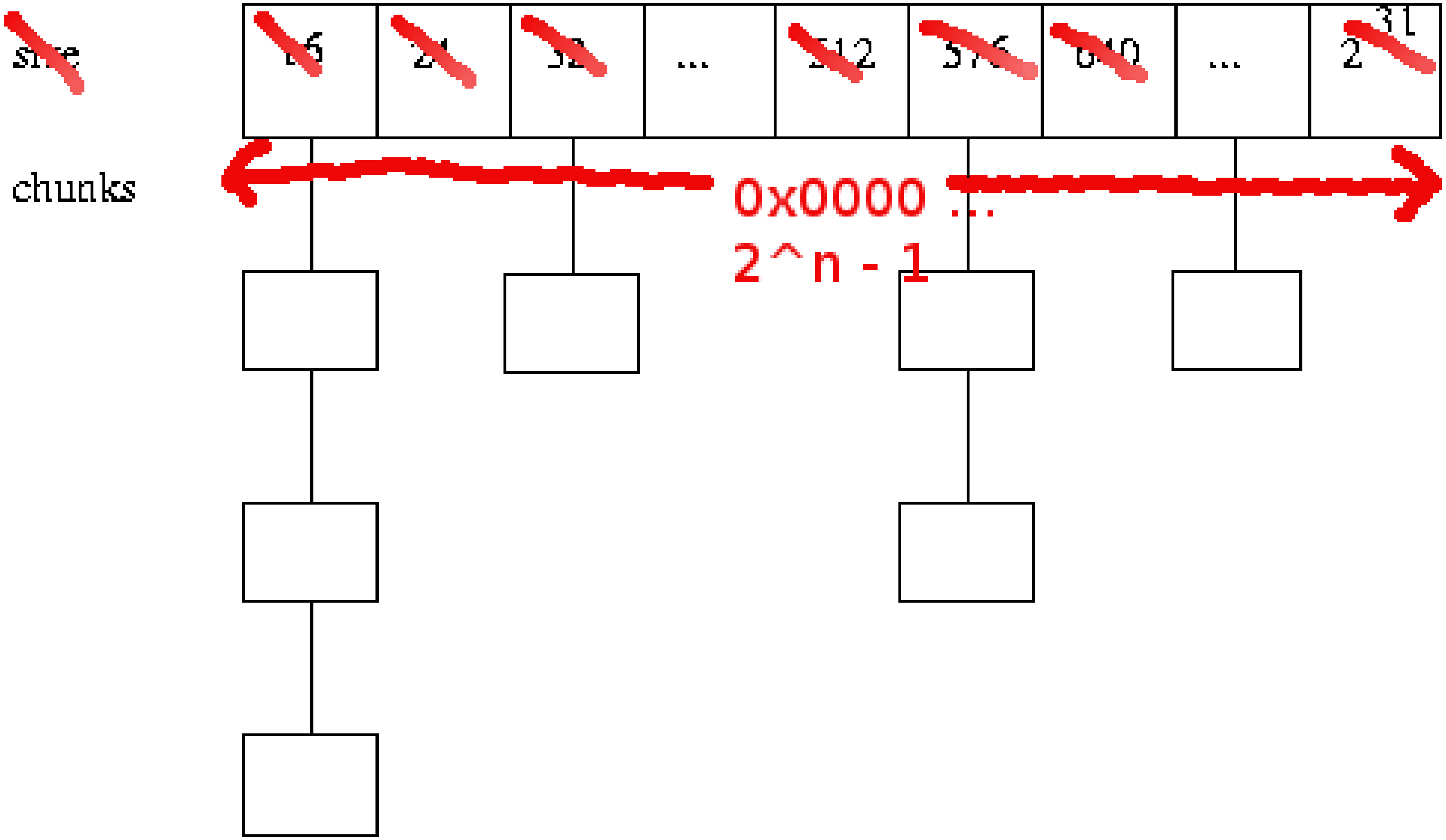
Indexing chunks

Inspired by free chunk binning in Doug Lea's (old) malloc.



Indexing chunks

Inspired by free chunk binning in Doug Lea's (old) malloc.



As well as indexing *free* chunks binned by *size*,
. . . index *allocated* chunks binned by *address*

How many bins?

Each bin is a linked list of chunks

- thread next/prev pointers through allocated chunks…
  - hook can add space, if no spare bits
- also store allocation site (key to DWARF info)
- can compress all this quite small (48 bits)

Q: How big should we make the bin index?
A: As big as we can!

- given an interior pointer, finding chunk is $O(binsize)$

Q: How big *can* we make the bin index?
A: Really really huge!

Really, how big?

Exploit

- sparseness of address space usage
- lazy memory commit on “modern OSes” (Linux)

Bin index resembles a linear page table.

After some tuning…

- 32-bit AS requires 2^{22} bytes of VAS for bin index
- covering n-bit AS requires 2^{n-10}-byte bin index…
- use bigger index for smaller expected bin size

What's the benefit?

Faster and more space-efficient than a hash table

- also better cache and demand-paging behaviour?

Some preliminary figures (timed gcc, 3 runs):

- gcc uninstrumented: 1.70, 1.76, 1.72
- gcc + no-op hooks: 1.73, 1.76, 1.72
- gcc + vgHash index: 1.83, 1.82, 1.85
- gcc + memtable index: 1.77, 1.78, 1.77

Memtables are not limited to this application!

- e.g. Cake "corresponding objects" look-up
- ... your idea here

Status of DwarfPython

Done: first-pass simplified implementation

- DWARF-based foreign function access
- no dynamic lang. features, debugger support, ...

Full implementation in progress...

- including proof-of-concept extension of LLDB
- + feedback into DWARF standards!

Tenuous link. . .

What's the big picture behind DwarfPython?

- habilitation of new / dynamic / unusual languages
- . . . into a mainstream toolchain
- language-independent notion of "API"
- orthogonalise language from tool support

What other neat tools might now be applicable to Python?

- tracers (e.g. ltrace)
- race detectors (helgrind or similar)
- heap profilers (massif, . . .)

What about verification / bug-finding tools?

Very quick summary

Wanted: a tool that can answer questions of the form:

- “how does my program exercise this API?” (general)
- e.g. “how does my program use the filesystem API?”
  - what data will it write? delete/overwrite?
  - what data will it *not* write? lose on crash?

How? Using Klee, a “dynamic symbolic execution” engine.

- works on binaries (LLVM bitcode as it happens)
- is it a static or a dynamic analysis? Hmm!

Ask me for more about this…

Conclusions & work in progress

Language implementors can do more to

- make using foreign code easier;
- orthogonalise language from tool support.

Questions for the audience:

- pessimal cases / bad GC interactions?
- can we do better?
- other uses of memtables? ("less conservative" GC?)

Still to do: implementation, benchmarks. . .

Thanks for listening. Any questions?

Taster: wrapper-free FFI (2)

Calling native functions:

- instantiate the data types the function expects
- call using `libffi`

In Parathon, an earlier effort, we had:

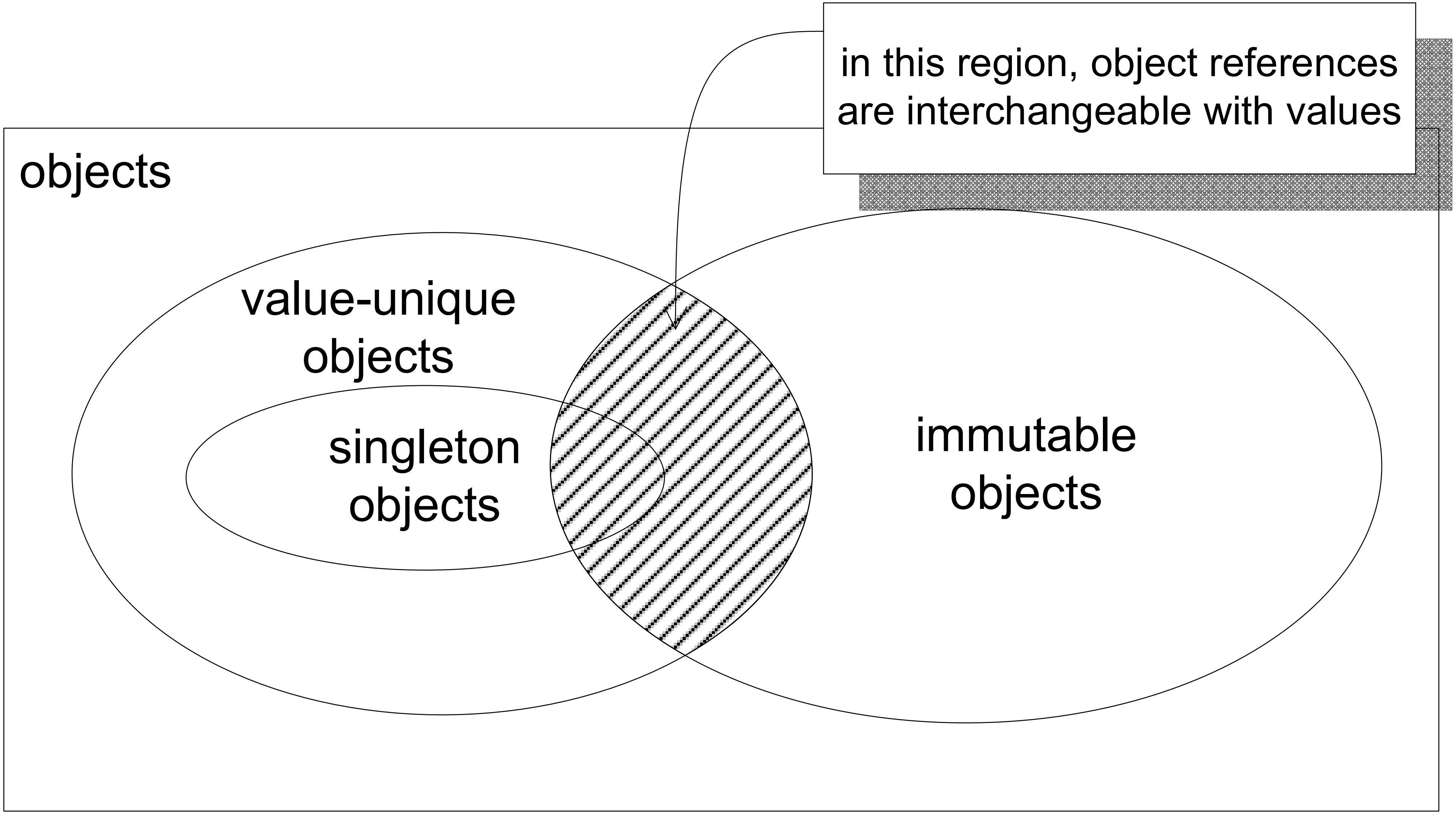
```
ParathonValue* FunctionCall::evaluate(ParathonContext& c)
{   return call_function (this→base_phrase→evaluate(c),
    /* invokes libffi ^ */   this→parameter_list→asArgs(c));   }
```

Now we have:

```
val FunctionCall::evaluate()  // ←— only context is the *process* i.e. stack
{   return call_function (this→base_phrase→evaluate(),
                          this→parameter_list→asArgs());   }
```

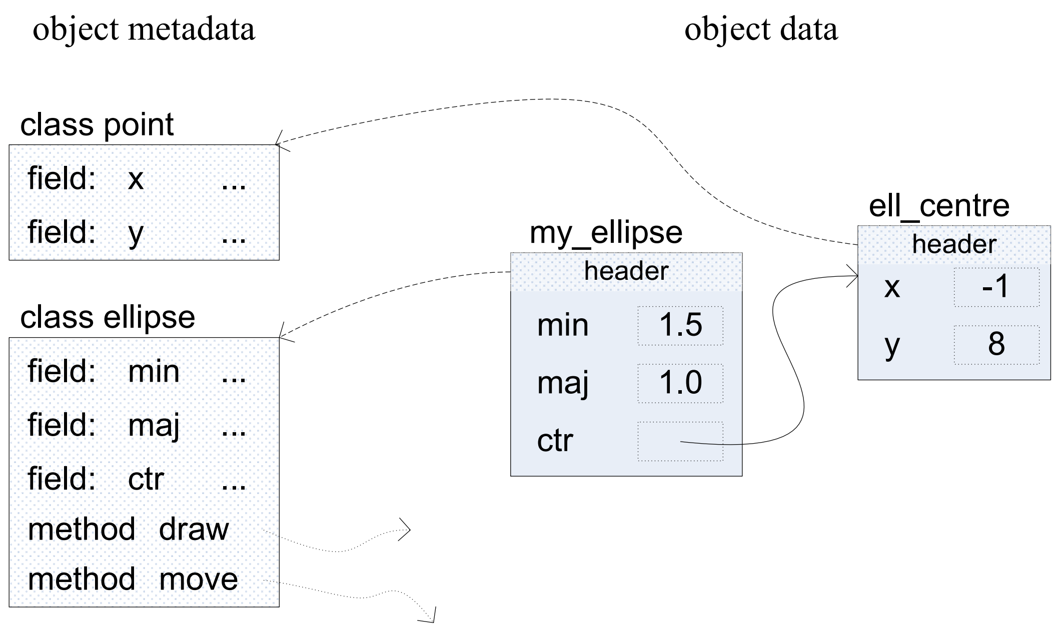
The interpreter context *is* the process context!

Primitive values



Out-of-band metadata

traditional approach: in-band headers point to object metadata



DwarfPython approach: metadata kept out-of-band and looked up associatively