HotPy (2)

Binary Compatible
High Performance
VM for Python

Mark Shannon

Who am I?

- Mark Shannon
- PhD thesis on building VMs for dynamic languages
- During my PhD I developed:
  - GVMT. A virtual machine tool kit including a JIT-compiler generator.
  - HotPy (1) A Python VM build with the GVMT.
- I'm currently looking for work in the UK

HotPy (2)

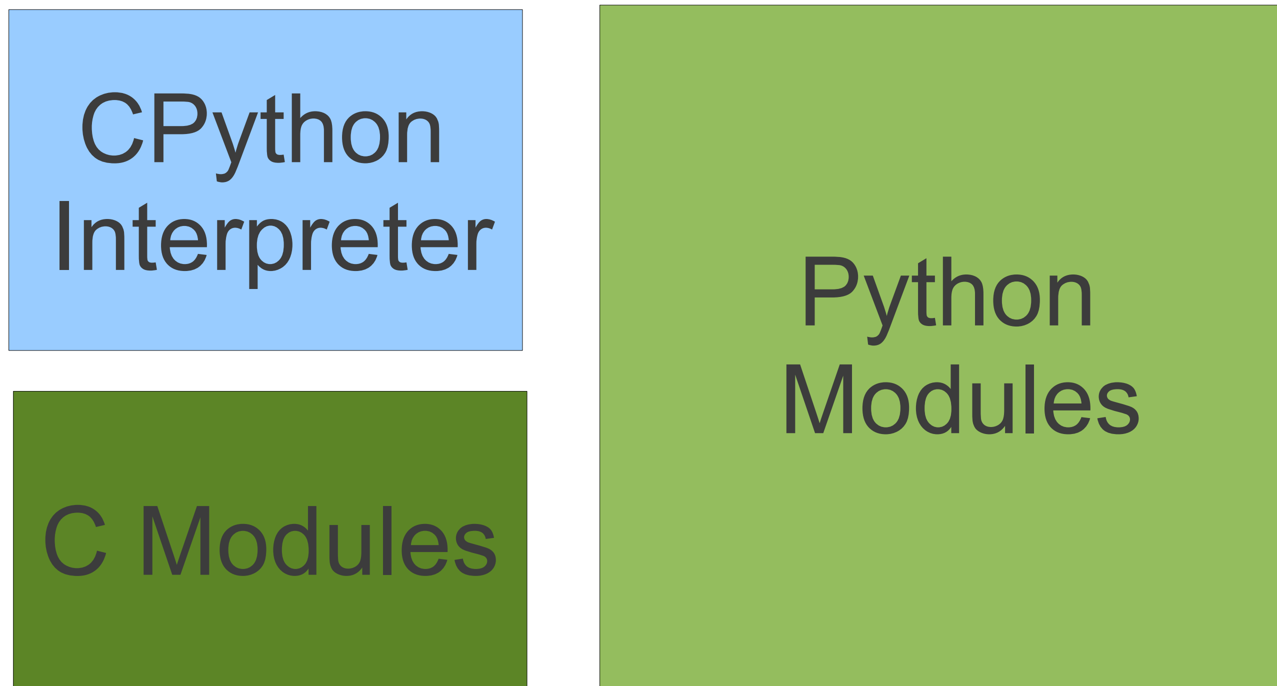
- HotPy (2) is:
  - a branch of CPython (3.3)
  - a tracing optimising *interpreter*
  - binary compatible with CPython
  - Currently about the same speed as CPython
  - Expected to be faster quite soon.

Why Bother with HotPy (2)?
(Why not just use PyPy?)

- Binary Compatibility and Portability
- CPython has many C extensions
- CPython can be embedded in other applications
- CPython runs on many platforms
- But CPython is slow(er than PyPy)

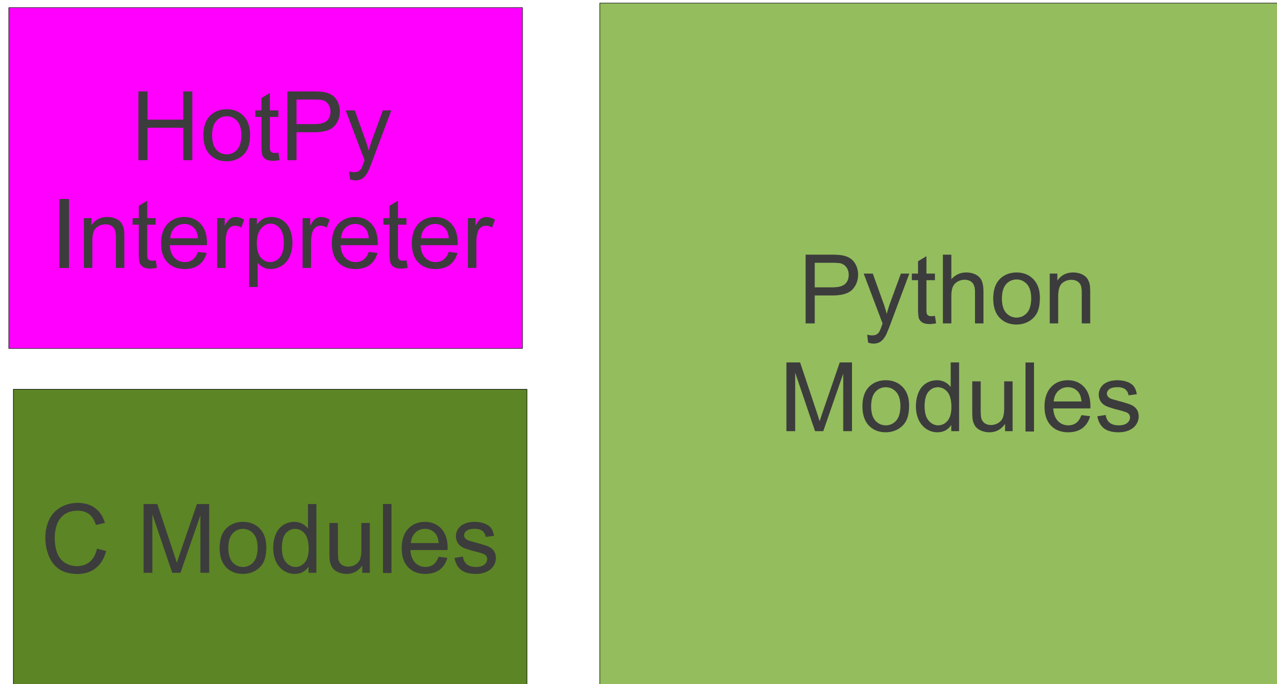
Binary Compatibility (Informally)

- Drop in replacement for CPython
- Replace CPython executable with HotPy executable and it will just work.



Binary Compatibility (Informally)

- Drop in replacement for CPython
- Replace CPython executable with HotPy executable and it will just work.



Binary Compatibility (More Formally)

- Full function API
  - including those with leading underscore
- ABI compatibility of PEP 384 (at a minimum)
- Layout of the PyThreadState and PyFrameObject will change.
- A few C extensions may need a little work
  - Particularly code that is very tightly bound to CPython such as Cython.

High Performance

- For pure Python code expectation is x2 faster
- When (if) compilation is implemented x5-8
- Dynamic optimisation removes much of the overhead of CPython.
- Techniques from all over the place, some my own, some pioneered by PyPy, many from the academic literature.

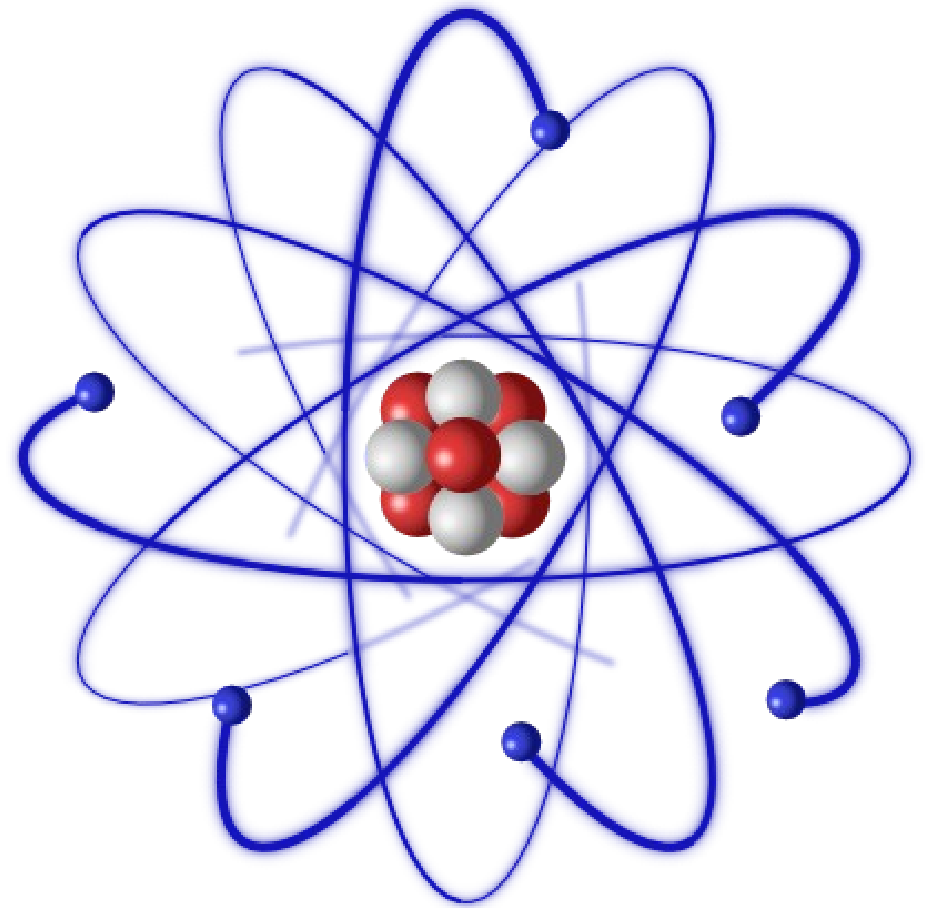
Optimising the CPython VM

An ideal model of the CPython VM.

- The *entire* state of the VM should be resident in heap memory in between bytecodes
    - The state of the VM should be fully described by the in-memory data structures
    - No part of the VM state should be embedded in the C stack or kept in a register.
- Each bytecode should be atomic
    - i.e. it should be impossible to observe the state of the VM part way through its execution.

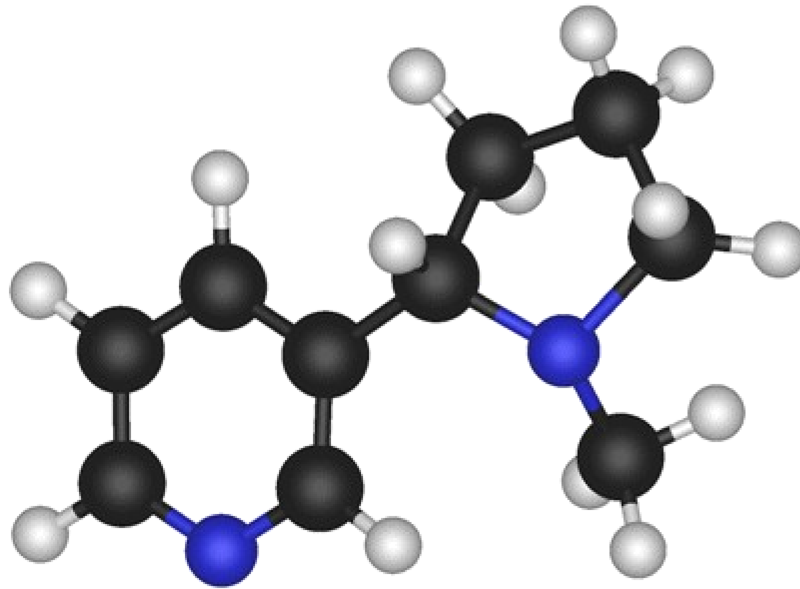
Atomic Bytecodes.

- An atomic bytecode cannot be observed part way through its execution
- Any calls made during the bytecode must be tail calls.



The Real CPython VM

- Lots of “micro” optimisations
- VM state is embedded in the C stack
- Many bytecodes are compound (not atomic)



Making the CPython VM Optimisable

- Before optimising CPython, we must make it optimisable
- All bytecodes must be atomic
- *Or* have equivalent function that is implemented in terms of atomic bytecodes
- Full VM state must be resident in heap memory
  - About 10% slower.

Making CPython bytecodes atomic

- Have to add (about 40) new bytecodes
- Each performs an atomic operation
- These are not hardware-level operations
- Can perform dictionary access or other intermediate level operation
- All can be completely described in terms of their effect on VM state.

Low-level Python

```
def surrogate_type_call(cls, args, kws):
    obj = cls.__new__(cls, *args, **kws)
    if $subtype($type(obj), cls):
        res = obj$__init__(*args, **kws)
        if res is not None:
            raise TypeError("__init__() ”
               “should return None not ”
               “'%s'" % $type(res)$__name__)
    return obj
```

Low-level Python

- `$op()` represents a single bytecode
  - Not a load of “op” followed by a function call
  - E.g. `$type(x)` is the type of x, not a call to type()
- `obj$attr` represents the LOAD_SPECIAL bytecode
  - Like load-attr but uses the “special” attribute lookup used by Python for methods such as __add__
- The LOAD_SPECIAL bytecode is a compound

Low-level Python

```
def surrogate_load_special(obj, name):
    cls = $type(obj)
    if $has_class_attr(cls, name): # Has class attr
        descriptor = $get_class_attr(cls, name) # Get class attr
        desc_type = $type(descriptor)
        if $has_class_attr(desc_type, '__get__'):
            getter = $get_class_attr(desc_type, '__get__')
            if $type(getter) is FunctionType:
                return getter(descriptor, obj, cls)
            elif desc_type is PropertyType:
                return descriptor$fget(obj)
            else:
                return $descriptor_get(descriptor, obj)
        else:
            return descriptor
    else:
        msg = "'%s' object has no attribute '%s'" % (
                cls$__name__, name)
        raise AttributeError(msg)
```

Optimisation in HotPy

- Concentrate efforts on “Hot Spots”
- Dynamically customise the code
- Avoid doing any work that can be avoided
  - Focus more on not doing things at all
  - Don't worry too much about doing them faster

Customisation (1)

- Assume that next time a piece of code executes, it will do so in a "similar" environment to this time.
- Tailor the code for that environment.
- By "environment" we mean the types of variables, the values of global variables holding classes and functions, the direction of branches taken and a few other things

Customisation (2)

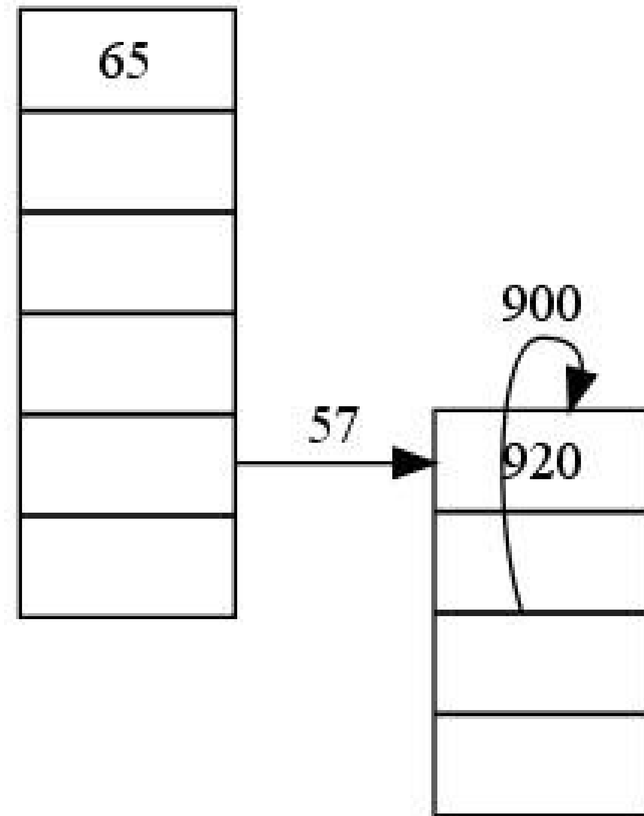
- Tracing
  - Customisation by flow. Record traces of commonly executed paths in the program.
- Specialisation
  - Replace general instructions with specialised versions
- Specialisation can done by tracing
  - E.g. Tracing a branch through a type test is effectively specialisation.

Traces and Tracing

- Follow actual execution, ignoring the structure of the program, focuses on the parts of the program that actually matter.
- Technique of choice for optimising dynamic languages
- Used by PyPy and several Javascript engines

Tracing Example (1)

```
def fact(n):
        x = 1
        while n > 1:
              x *= n
              n -= 1
        return x
```



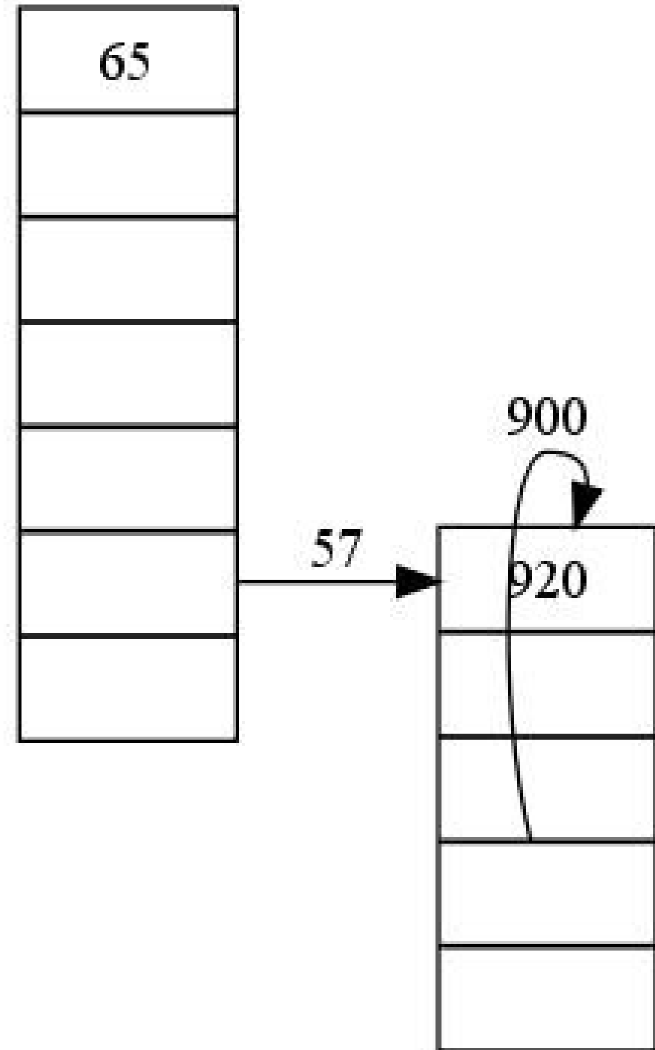

```
fact(1000)
import hotpy.trace
hotpy.trace.graph_snapshot(
'/tmp/graph1.gv')
```

Tracing Example (2)

```
def mul(a, b): return a * b

def fact(n):
    x = 1
    while n > 1:
        x = mul(x, n)
        n -= 1
    return x

fact(1000)
import hotpy.trace
hotpy.trace.graph_snapshot(
'/tmp/graph2.gv')
```



Traces (Output of Tracing)

- Calls to functions (and returns) are inlined into the trace
- Traces are linear and are simpler to optimise than the complex flow-graphs of functions
- Traces are usually longer than functions
  - Exposes more optimisation potential

Guards

- During traces some assumptions are made
  - Branch instructions always go the same way
  - Call sites always call the same function
- Some of these assumptions will prove to be false so "guards" must be inserted to ensure correctness
- Guards check that these assumptions are correct.
- If assumption is false then the trace exits.

Trace Management

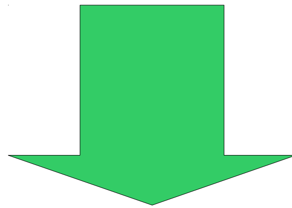
- When a backwards edge becomes warm:
  - Trace it and add it to trace cache
- Decay execution counts over time
  - Traces no longer used will then become “cold”
- When a trace becomes cold:
  - Mark trace as invalid and discard.
  - Limits to trace cache to ~1M (with current settings)

Specialisation of Traces

- Some specialisation of traces is a consequence of generating the trace.
  - Can only record the taken branch of an if statement
  - Need to customisation on the *value* of functions in order to trace calls to them.
- Further customise by *specialising* by type
  - Replace general bytecode with type-specific one(s)
  - Type-specific operation are often a lot faster and can often be no-ops

Specialisation Example

1. ENSURE TYPE x, slice, exit12
2. t0 = TYPE(x)
3. t1 = t0 IS slice
4. EXIT_IF_FALSE t1, exit13



1. ENSURE TYPE x, slice, exit12
2. t0 = slice
3. t1 = True
4. NO-OP

Redundancy Elimination

- Customisation exposes much of the redundancy in the Python execution
  - Customisation may add some extra redundancy
- Deferred Object Creation:
  - Lazily create intermediate values.
  - Often these object do not need to be created at all.
- Register Interpreter
  - Remove redundant moves to and from the stack.

Deferred Object Creation (1)

- Split VM state into two parts
  1. Real in-memory data structures
  2. A “recipe” for reconstructing the remaining part
- The “recipe” is a constant over time for any given location on a trace.
- Only need to generate the “recipe” once

Deferred Object Creation (2)

- Split the VM state so that as much of the delta in the VM state from one bytecode to the next is incorporated into the "recipe" and as little as possible is incorporated into the in-memory part.
- The changes in the in-memory part represent real work that the VM must do. Changes to the recipe have no runtime cost (unless we are forced to use the recipe).

Deferred Object Creation (3)

- For each (low-level) bytecode:
  - Modify the recipe to reflect how to recreate the VM.
- If any objects are needed to perform an op:
  - Create those objects from the recipe
  - Generate the code to do the work
- If bytecode may exit the trace
  - Record the recipe at this point
  - If no exit is possible, there is no need to record the recipe

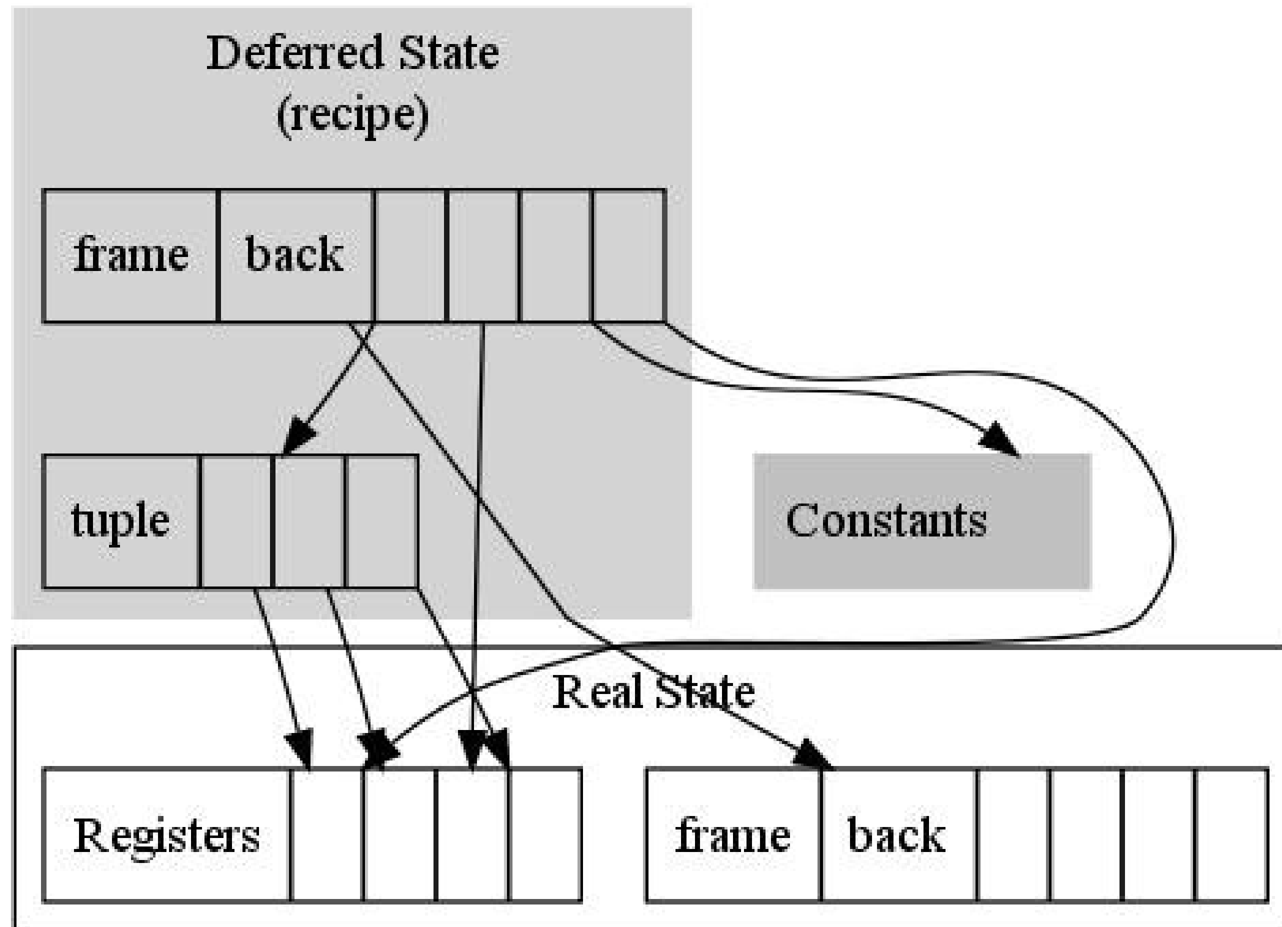
Deferring Objects

- Objects have *values* as well as shape
- These values must be stored somewhere
  - Store them in registers
- Storing values in registers allows frames to be deferred.
- Deferring frames is probably the most powerful optimisation employed by HotPy.

Deferring Example

- Deferring creation of a tuple:
  - Bytecode BUILD_TUPLE 3
  - Record in recipe:
  - That the stack is now two shallower
  - That the top of the stack is a tuple consisting of the previous 3 items on the stack

A pictorial recipe

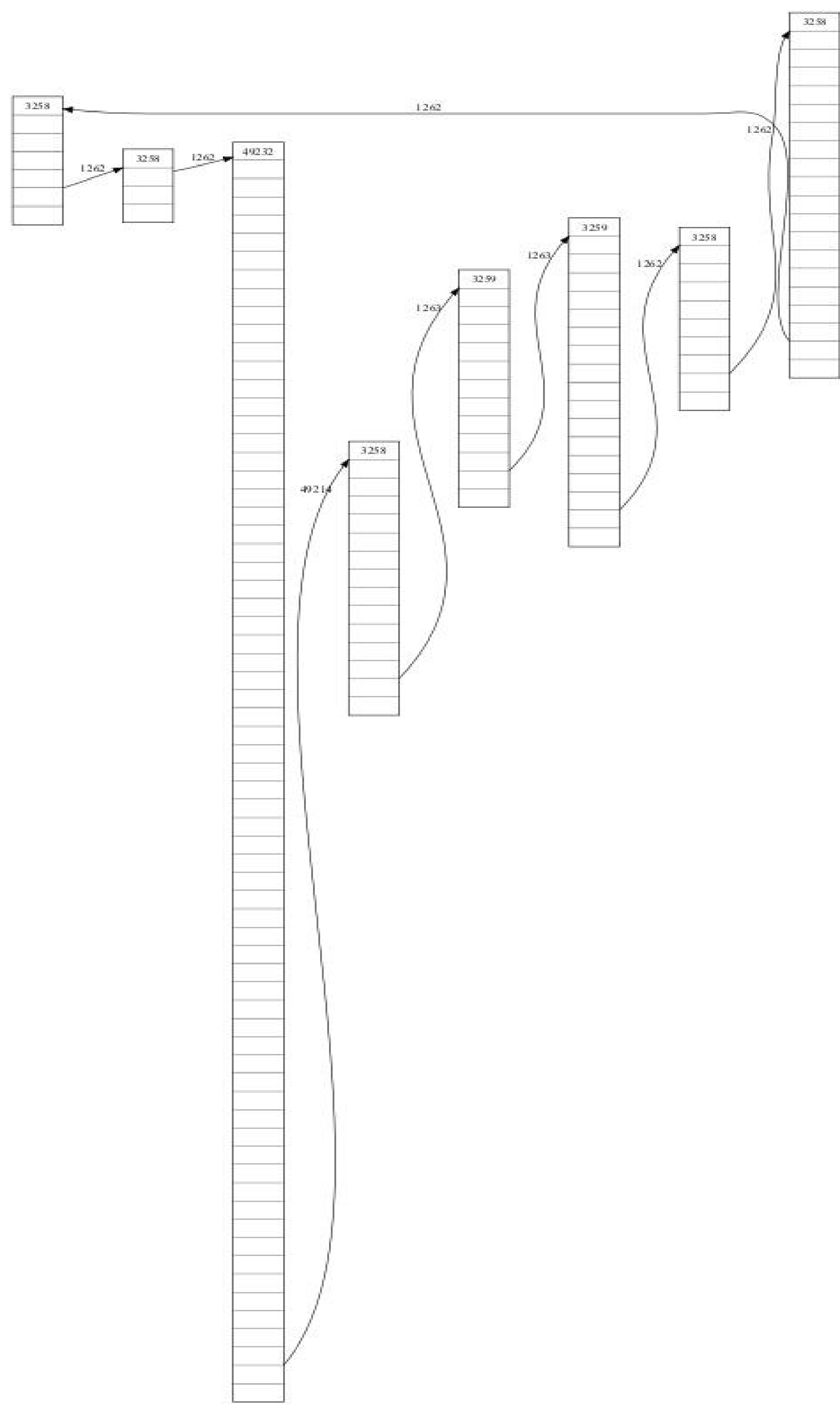


Example of Optimisation

- pystones
  - Everyone's favourite benchmark :)
- Try it yourself
  - python -m test.pystone
- HotPy options to try out:

  -Xspecialise -Xtailcall

  -Xdefer -Xregister -Xcoalesce



3258
1262
3258
1262
1262
49232
3258
1262
1263
3259
1263
3259
1262
3258
49214
3258

Instruction Counts

- No tracing: 24.7 million instructions
- Tracing (no optimisations): 460 million!
  - Many more low-level bytecodes are required
- With Specialisation: 456 million
  - Many of these would be trivial remove
- Specialise and DOC: 91 million
    - A lot of stack→reg & reg→stack moves
- Specialise, DOC, Register: 23.1 million

Speed

- Not many benchmarks as most of the optimisers are not very robust yet :(
- Generally about as fast as CPython
  - Pystones 25% faster on my netbook
- Lots of optimisations yet to do:
  - Dictionary based optimisations
  - Unboxing
  - An infinity of other things

Conclusions

- HotPy needs to be fast *and* very reliable
- Currently, HotPy is fast *or* quite reliable
- If you want a faster Python and can't use PyPy then you need to help develop HotPy:
    - Development and patches
    - Set up buildbots
    - Fix speed.python.org and add HotPy
    - Donations
    - Any other help you can offer

Thank you listening

Any questions

www.hotpy.org