Comparing the performance between a stack-based VM and a register-based VM

… and the results

Alex Fang <frjalex@temple.edu>

The Question

- Performance wars between interpreted languages
- E.g. Python vs. Lua, JVM vs. Erlang VM, etc.
- If one of them, e.g. JVM, is faster, then **WHY**?
- The performance of an interpreted language is essentially the virtual machine’s performance

Into the virtual machine

- Two MOST popular types of VM architectures today:
- Stack-based architecture: CPython, JVM. Employs stacks to store data; implicit to specify memory address of storage in byte code
- Register-based architecture: Lua, PHP*. Simulates a physical CPU (operates on registers; memory address needs to be explicitly specified)
- Other architectures exist, e.g. Hybrid, Accumulator (not extremely popular)
- *: Another controversial topic

Stack-based and register-based VM, **which is faster?**

Problem rephrased

- 1: What approach to take?
- 2: Measuring performance
  - The less time it takes, the faster!
  - **dispatch time** = the time spent matching a byte code instructor with a particular operation function on VM (we use only switch dispatch);
  - **fetch time** = the time spent to fetch ONE instruction from a sequence of processed inputs
  - **execution time** = total time spent executing byte code instructions

What is performance?

- In an interpreted language: runtime performance
- Directly, the less time the better
- Since register byte code contains 3 operands and stack contains only one, stack can be faster on overall fetch time
- Since register byte code contains memory addresses (and stack doesn't) registers can be faster on dispatch time, having less amount of dispatches
- Factors of performance:
  - Amount of dispatches
  - Total time spent in dispatch
  - Total time spent in fetch
  - Execution time (not including the part parsing byte code)

Our approach

- Writing two new VMs with timing mechanisms built-in and with structural similarities in mind - and measure the runtime performance - “Conceptum”, the stack-based VM, and “Inertia”, the register-based VM, written entirely in ANSI C11

- Timing: the less the faster

- A few vocab:

- dispatch time = the time spent matching a byte code instructor with a particular operation function on VM (we use only switch dispatch);

- fetch time = the time spent to fetch ONE instruction from a sequence of processed inputs

- execution time = total time spent executing byte code instructions

Result: Dispatch amount

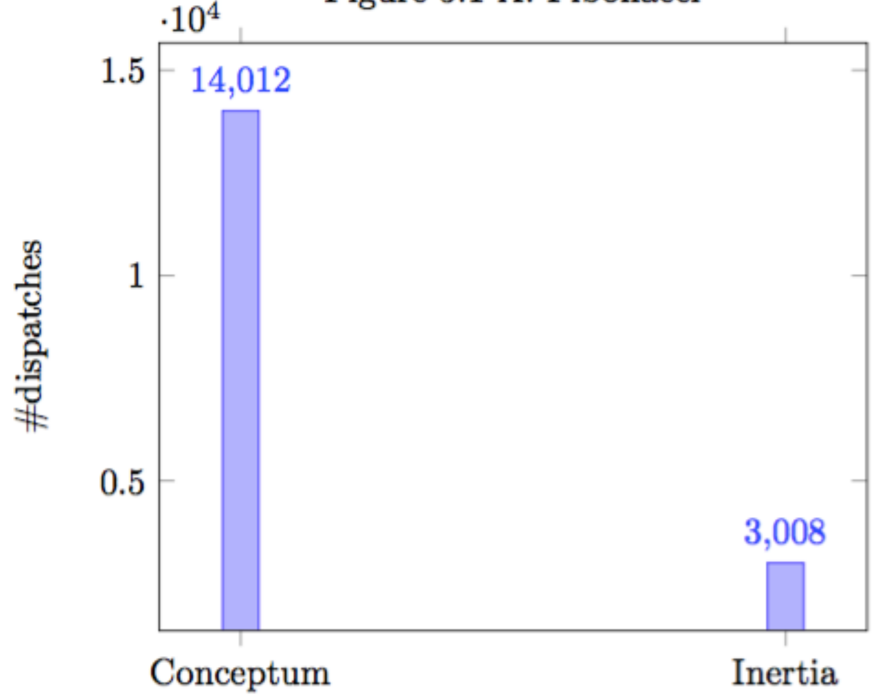


Figure 5.1-A: Fibonacci

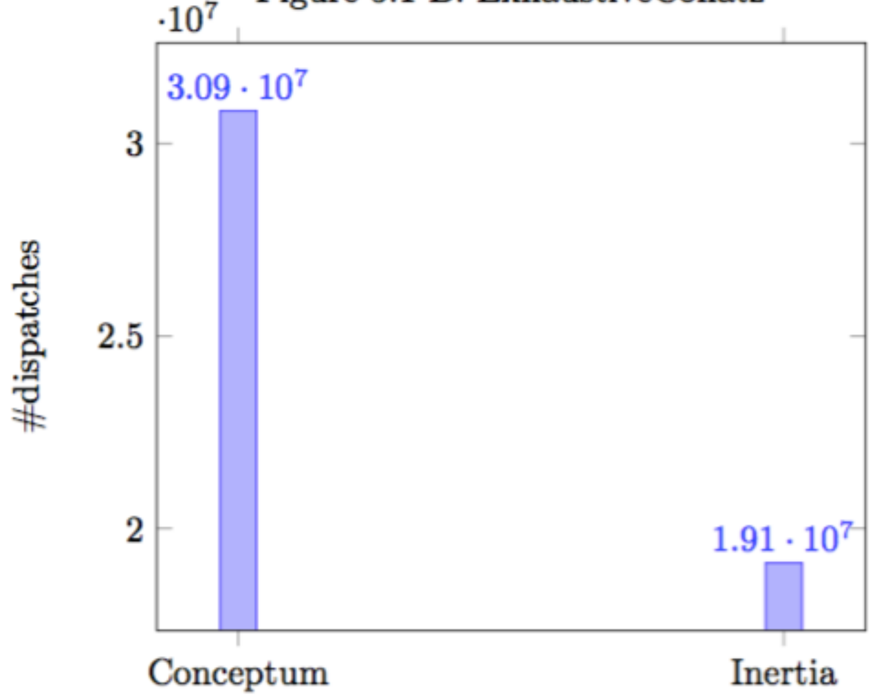


Figure 5.1-B: ExhaustiveCollatz

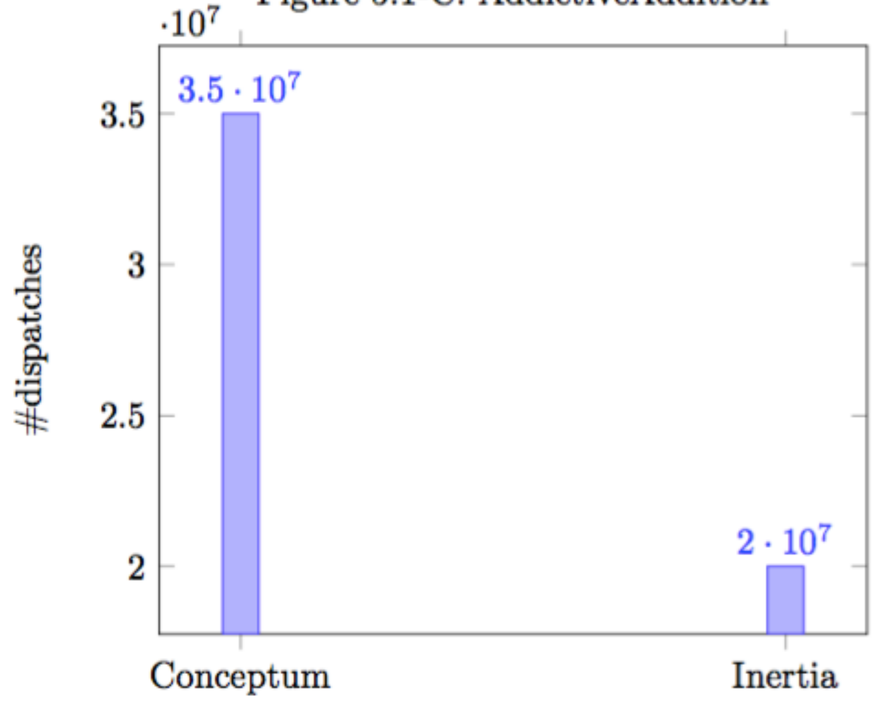


Figure 5.1-C: AddictiveAddition

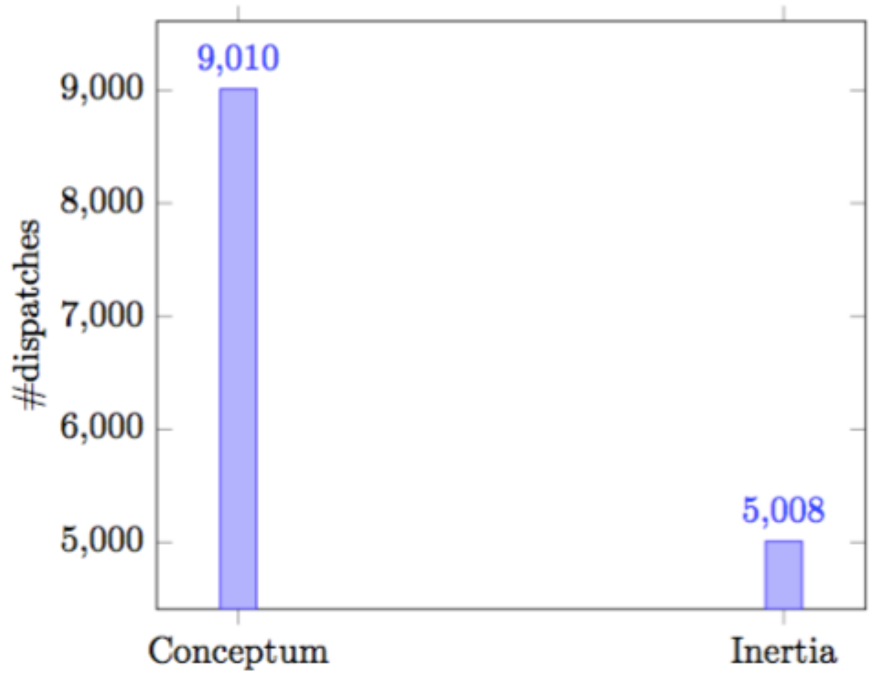


Figure 5.1-D: Recursion

Result: Time spent

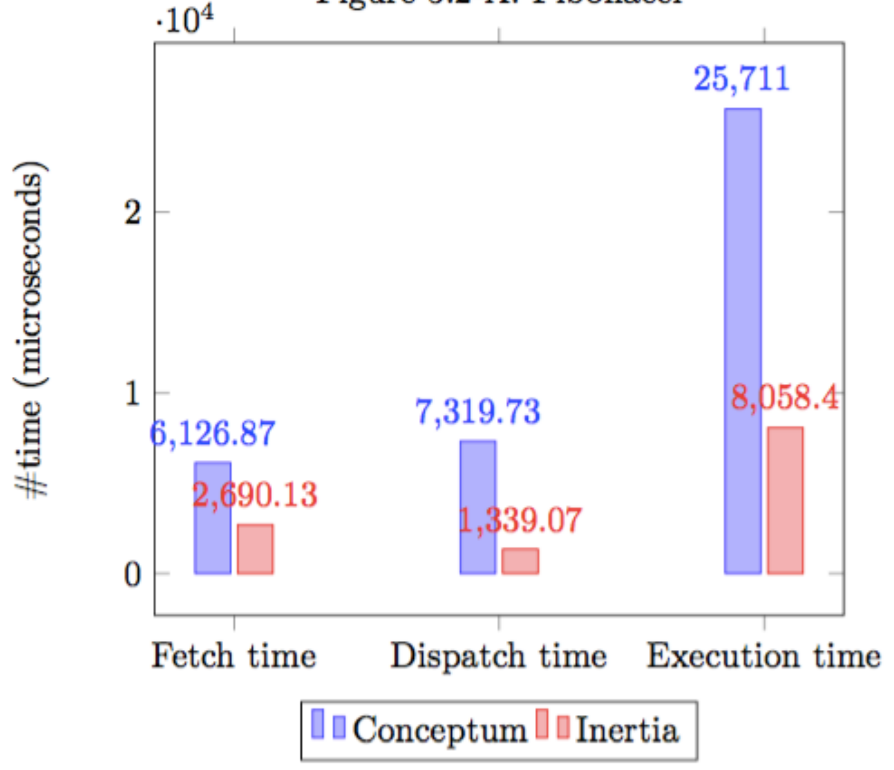


Figure 5.2-A: Fibonacci

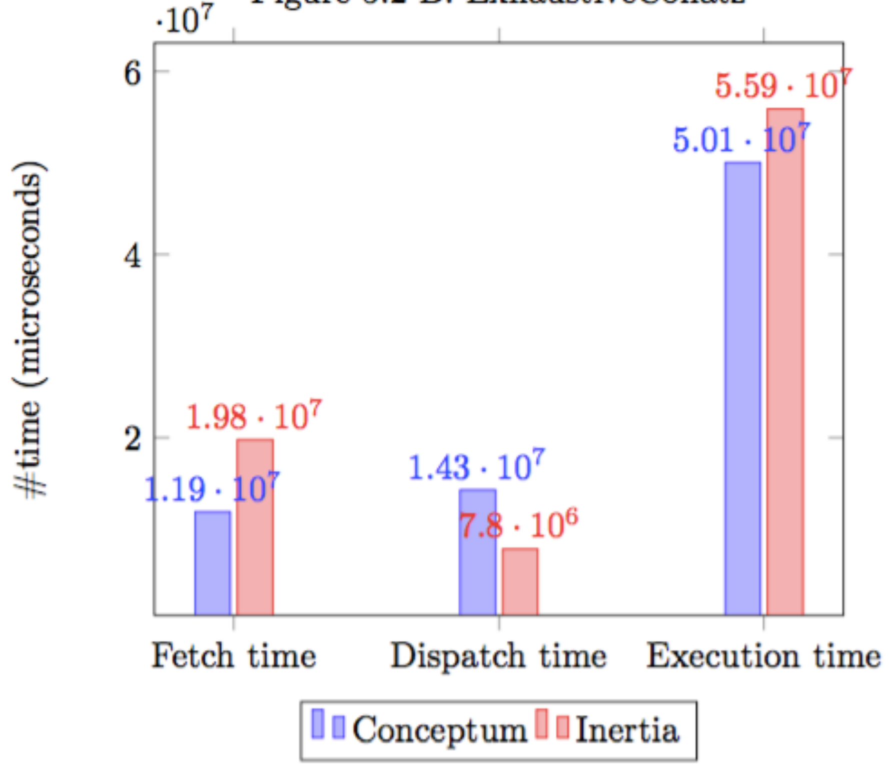


Figure 5.2-B: ExhaustiveCollatz

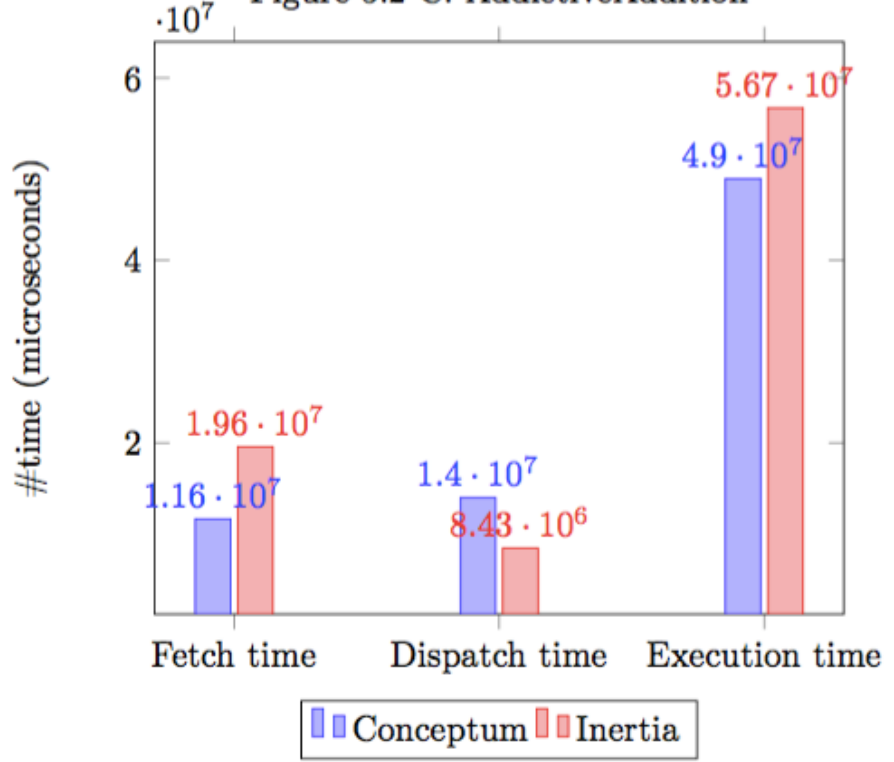


Figure 5.2-C: AddictiveAddition

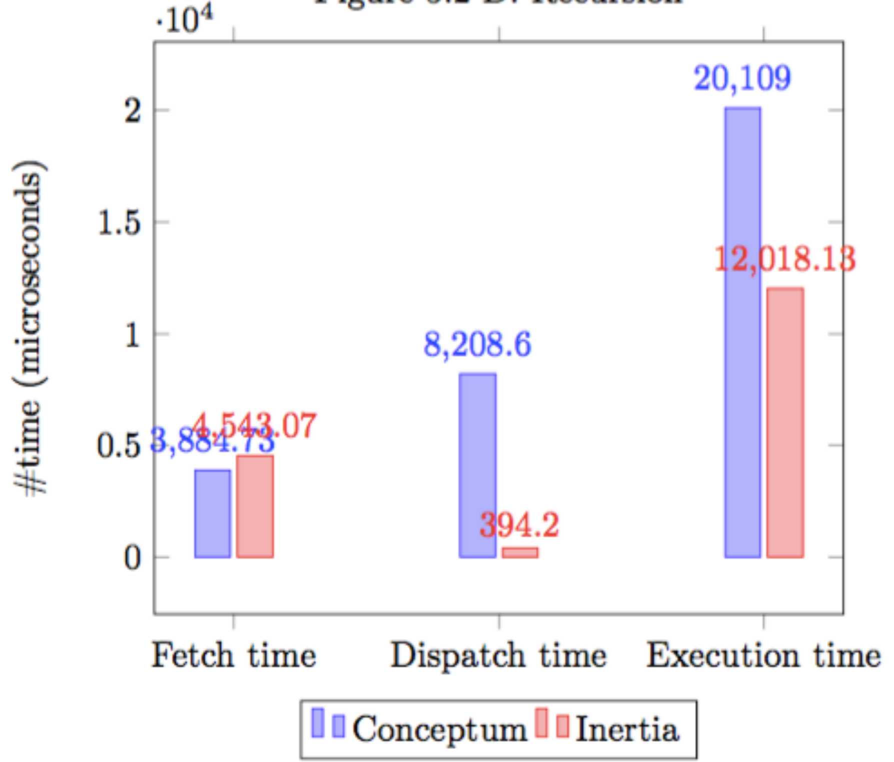


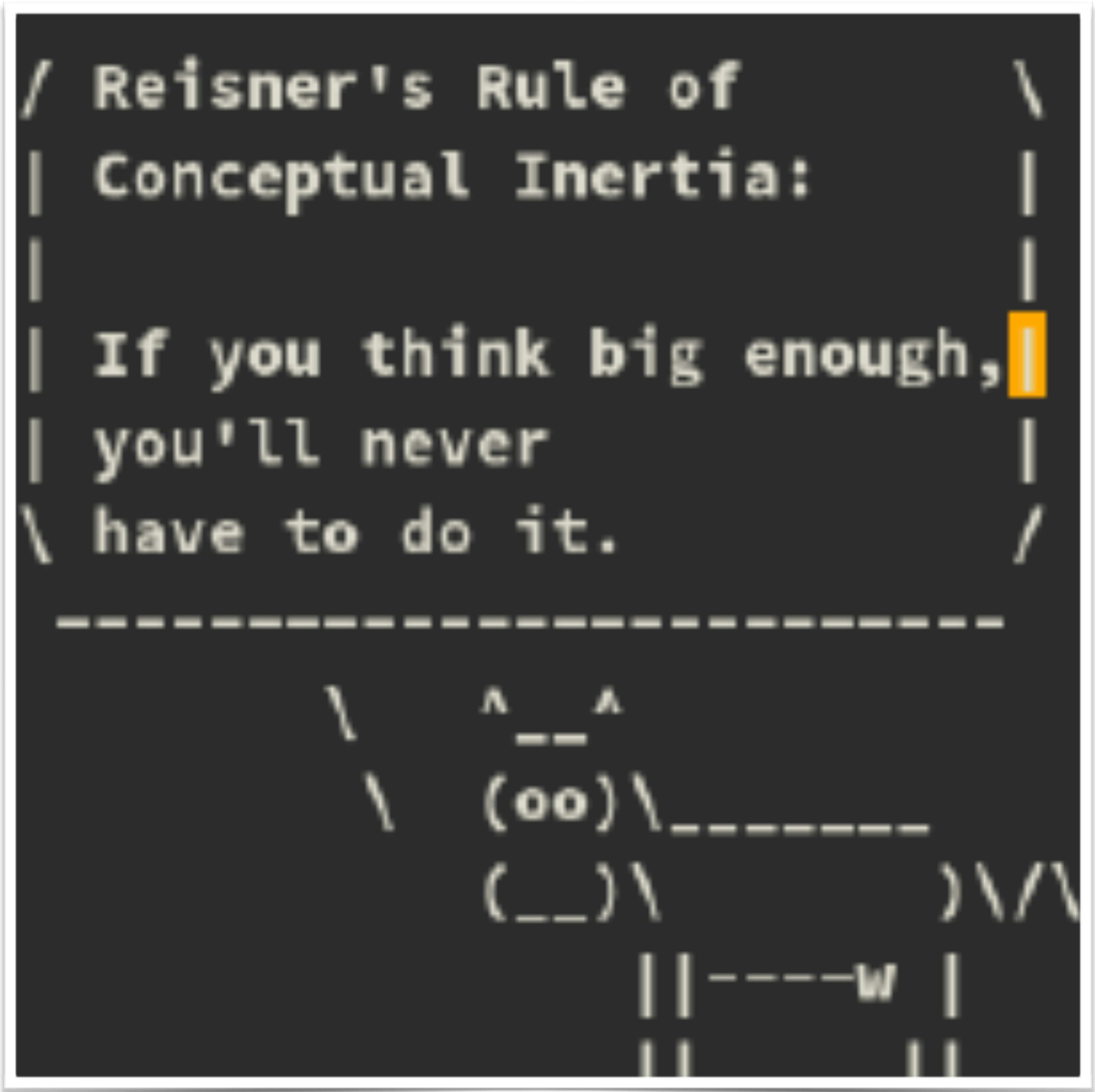
Figure 5.2-D: Recursion

Conclusion

- Overall a register-based VM implementation is faster
- Stack-based VM performs better in fetch time (less)
- If you want to implement a high performance, compact DSL on limited hardware, go for a register-based VM!
- If you favor simplicity (both in byte code and in code for VM) over performance and want to perform dense read/write to the VM's memory space, implement a stack-based VM!

Source code?

- This PDF available at https://www.github.com/Conceptual-Inertia/presentations/plugtalk.pdf
- Source code of Conceptum available at: https://www.github.com/Conceptual-Inertia/Conceptum
- Source code of Inertia available at: https://www.github.com/Conceptual-Inertia/Inertia
- The official paper available at: http://fat-sausage.derros.in/papers/vmplug.pdf
- Questions? Critics? Suggestions? Email: frjalex@temple.edu



```
/ Reisner's Rule of          \
| Conceptual Inertia:        |
|                            |
| If you think big enough,   |
| you'll never               |
\ have to do it.             /
 -------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```