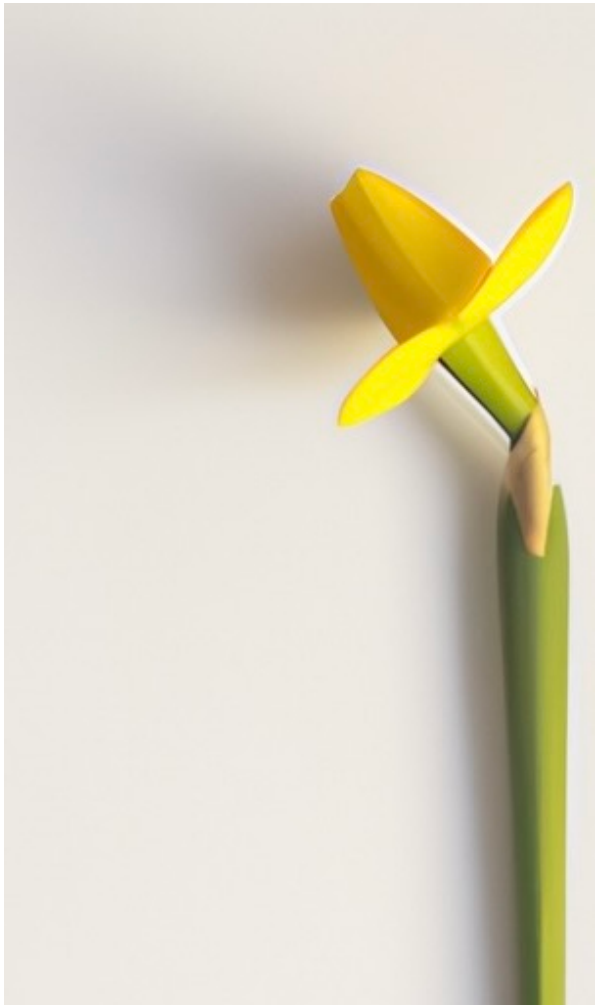
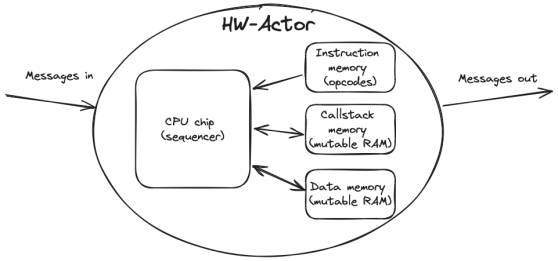
HW Actors



Actor Ideas Pre-Date Hewitt's Formalization

MOP Is The Future



CPUs Are Actors

- Shared registers
- Shared stack, stack pointer
- Shared instruction pointer
- Shared mutable store - RAM, registers
- Sequential, step-wise interpretation of opcodes
- Hardware originally intended to run single-threaded software
  - CPUs were very expensive in 1950s, hence, human-time was spent to fake out multiple CPUs (also known as “time-sharing”)
- Control-flow - opcodes - are an interpretation of data stored in memory
- Other data in memory is not interpreted by hardware
  - Non-code memory interpreted and given “meaning” by software (“apps” rather than hardware)
- CPUs cannot share memory with other CPUs, except across wires
  - Software model of “multi-tasking” included unrealistic concept of shared memory between fake CPUs, causing plethora of accidental complexities
  - Accidental complexity of conflating mathematical *functions* with CPU *subroutines* was resolved in an ad-hoc manner using *preemption* and the invention of *process-based operating systems*

Modern Hardware Actors

- Very different from 1950s hardware
- Lots of memory
- No need for time-sharing
- Inexpensive CPUs
- No need for bloated process-based operating systems
- Yet, we continue to use 1950s techniques for programming this new kind of hardware

Arduino, Raspbery Pi, etc.

- Inexpensive, modern Actors
- Enormous amounts of RAM/ROM

CPUs Are Easy-To-Alter Hardware Devices

- The idea of re-programming machines has been around for a long time
  - But, was slow and inconvenient
  - e.g. machining new metal gears for physical machines
  - Unsoldering ICs and wires from circuits
  - Etc.
- The idea of attaching some RAM/ROM to a sequencing circuit is central to the ideas of improving re-programmability of machines.

The Term ”Computer” Is Too Limiting

- “Compute-er” suggests that the *only* purpose of electronic machines is to create compute-ing devices, i.e. calculators
- Re-programmable Electronic Machines can do *more* than simply compute-ing results
  - e.g. sequencing machines
  - e.g. sequencing video
  - e.g. robotics
  - e.g. sequencing game software
  - Etc.
- REM - Re-programmable Electronic Machine
- REMics for the science of Re-programmable Electronic Machines
- Computistry for the science of *computation*

Cpu subroutines are not mathematical functions FP is only a convenient simplifying assumption, but only *one*. FP makes writing calculators easier, but, makes writing asynchronous code harder. Notation mismatch.

Programming Simplicity

https://guitarvydas.github.io/2020/12/25/The-ALGOL-Bottleneck.html

https://guitarvydas.github.io/2020/12/09/CALL-RETURN-Spaghetti.html

MOP is Message Oriented Programming

See Also

References *https://guitarvydas.github.io/2024/01/06/References.html*
Blog *https://guitarvydas.github.io/*
Blog *https://publish.obsidian.md/programmingsimplicity*
Videos *https://www.youtube.com/@programmingsimplicity2980*
[see playlist "programming simplicity"]
Discord *https://discord.gg/Jjx62ypR*
X *(Twitter) @paul_tarvydas*
More writing *(WIP): https://leanpub.com/u/paul-tarvydas*

The original mindset for the design of CPUs was that of creating Hardware Actors.

You bolt a sequencer chip (CPU) to some private memory (RAM and ROM) and you get an Actor, i.e. a piece of hardware that is an Actor. The sequencer reads instructions, sequentially, from program memory. Originally, program memory was just a part of the private memory, loosely defined and not fenced in. Hardware Actors are not able to share memory with other Hardware Actors. A Hardware Actor can communicate with other Hardware Actors *only* by sending electronic blips across wires (aka Messages).

In the 1950s CPUs and memory were prohibitively expensive. So, human-time was spent on efforts to fake out multiple actors on single - very expensive - Hardware Actors.

Right off the bat, a huge mistake was made - fake actors were allowed to do something that real Hardware Actors, could not do, i.e. share memory. The mistake led to an avalanche of accidental complexities and resulted in big blobs of bloatware called “operating systems”. The original mistake was compounded by making a new mistake, that of conflating mathematical functions with subroutines used by the sequencer devices.

Today, in 2024, we have much cheaper sequencer chips and much cheaper memory. We have off-the-shelf hardware actors sold under brand names such as “Arduino”, “Raspberry Pi”, etc. And, we have the software chops to write less bloatful operating systems’ processes, i.e. closures. And, we have the software chops to fake out wires - queues. We can go back to the original model.

Hardware Actors could be used to build calculators, but, hardware Actors could be used to build other kinds of devices, too, e.g. sequencers (which are not, fundamentally, calculators).

Function-based programming languages allow you to build only one kind of thing easily - calculators. If you want to build other kinds of devices, e.g. sequencers, you have to spend time developing workarounds to compensate for the use of the function-based paradigm.

Function-based programming languages allow you to program the innards of single Actors, but, make it inconvenient to program networks of Actors.

I have found that using the original model of Hardware Actors is a very convenient paradigm for creating re-programmable electronic devices. Instead of bolting unwanted shared memory and unwanted sequencing into every program, these become exceptions to the fundamental rule of Actors. If you need shared memory for “efficiency”, then you solve the problem in some way - maybe by using a function-based approach, but, not necessarily. If you need synchronization, you solve the problem in some way - research in networking resulted in a plethora of solutions to the problem of synchronization, without the need to bolt synchronization into the bowels of every program ever written.

This “new” model of fake Hardware Actors is like Chuck Moore’s Green Array hardware. You program the innards of Hardware Actors using some convenient language (in Moore’s case, Forth, in other cases Python, Rust, etc.) and you orchestrate a bunch of fake and hardware actors to make a complete system.

The future of programming includes distributed devices, like internet, robotics, GUIs, blockchain, gaming, etc. I believe that these issues can be more easily solved using the Hardware Actor model instead of the function-based model.

We have built the beginnings of a programming language - 0D - that allows you to create distributed systems using the hardware actor model, while still being able to use the function-based model when appropriate.

This language allows you to mix existing function-based languages to form solutions using a plethora of languages. Some believe that using more than one language is confusing and painful. I believe that such opinions are based on bad prior experiences with the function-based paradigm.

This language allows you, also, to break free of the textual, character-based paradigm.

Most existing character-based languages - like Rust, Python, Javascript, etc. - form programs using strict grids of non-overlapping bitmap cells which must be read and executed in a sequential manner (top-down, left-to-right in English-speaking domains). 0D allows cells to overlap and to be resized. 0D imposes no strict ordering / sequencing of program units.

Furthermore, 0D is based on the principal that every layer must make complete sense in a stand-alone manner. Many existing languages and code libraries do not satisfy this principal, in that they contain hidden dependencies that strongly couple units of code (functions, libraries) together in ways that make testing and debugging difficult. Most existing languages are built in a manner similar to that of building intricate Swiss Watches. In this paradigm, all parts of a program are intermeshed together like gears in an intricate wristwatch. Asynchronous operation of any sub-portions is not allowed. Systems built this way are brittle. For a system to work, all subparts must work. If one part fails, the whole system fails. For example if one tooth breaks off of one gear out of hundreds of gears, in an intricate watch, the whole wristwatch stops working correctly.

Compose larger systems by composing layers

Libraries, finally, become LEGO® blocks. Build and forget. Once a software component works, it remains working. A component has no dependencies on other components. Changes in components cannot affect other components. A program composed of independent components can still be affected by changes in its children components, but components cannot affect each other directly.

Composition of composite components is recursive. Composite components can contain other composite components and/or leaf components.

Composition of composite components is explicit and visible. I.E. not PUB/SUB, not self-modifying code.

Contracts - tables of inputs vs. outputs - are possible and keep things simple enough to allow rapid identification of components that violate their contracts.

Components can consume multiple inputs and produce multiple outputs over spans of time.

Operation of components is not fundamentally reversible, which is the way that Reality works.

- Concept of “time” is eschewed in function-based approach.
- Possible, but not necessary, to side-step the use of operating systems
- Function-based languages attack only a minor, less significant portion of the programming problem
- Memory sharing remains an ornery problem in 0D, but, is taken off the critical path for other kinds of programming
- Let’s program like it’s 2024 instead of 1950
-

- Lack of readability and readability kills freethinking.
- Options kill readability and readability.
  - We learned this lesson with the transition from assembly programming to high-level language programming
  - Early high-level languages put all arguments on the stack, which looked “inefficient” at the time (but, was fixed later)
- The lesson is normalize. KISS and optimize later. Don’t optimize prematurely.
- Unfortunately, most popular programming languages, like Python, Rust, Haskel, Javascript, etc., encourage Waterfall design. Attitude: “I know how this needs to work, I will just code it up efficiently on the assumption that my design is perfect and does not need to be iterated”.
- We haven’t risen above 1950s style programming with the result that innovations are few and far between
  - We need to upgrade PoP from PoP, before we can imagine better FoP. (Present-of-Programming, Past-of-Programming, Future-of-Programming, respectively)
- REM - Reprogrammable Electronic Machines, instead of “Compute-ers”.