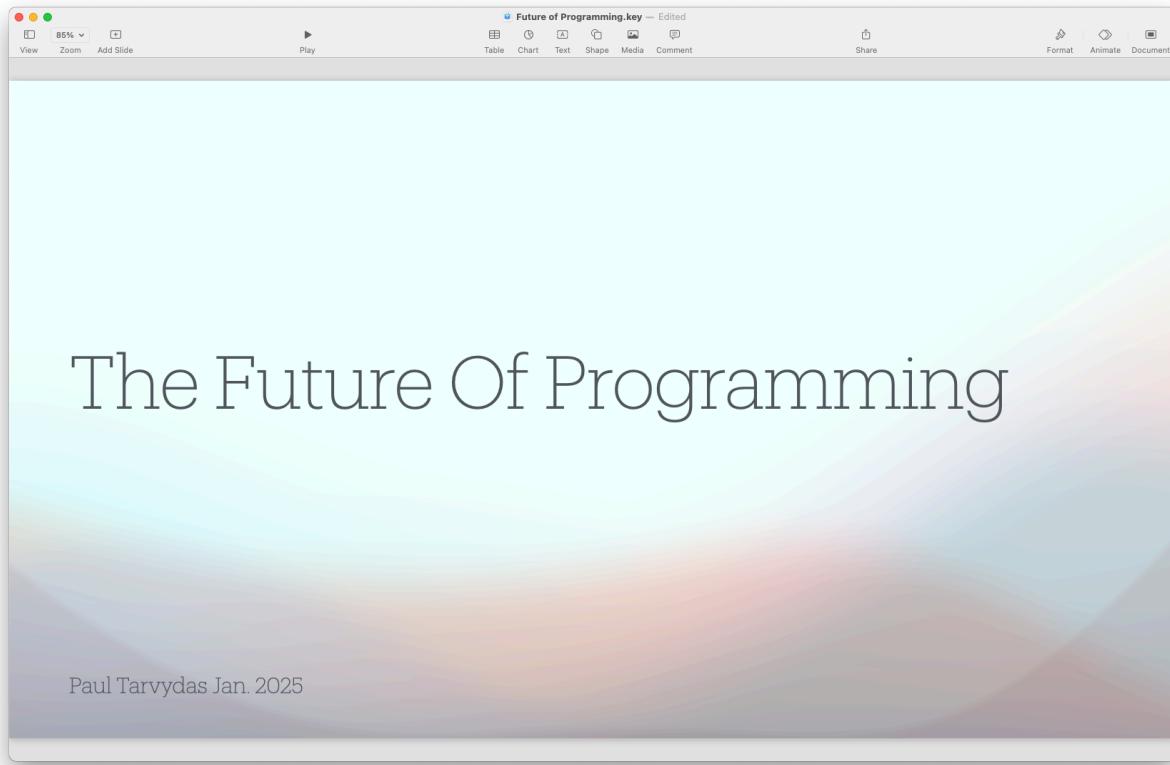


The Future of Programming

2025-01-02



Slides from YouTube video (see reference below)

Simple CPUs

Future of Programming.key

View Zoom Add Slide Play Table Chart Text Shape Media Comment Share Format Animate Document

Simple CPU

Breaking Tower of Complication

- CPUs were very expensive in 1950
- So CPUs were time-shared
- Memory-sharing was used by default (this led to “thread safety” becoming an issue in time-shared systems)

compute-ing, context-switching, virtual memory

Software

Hardware

1950

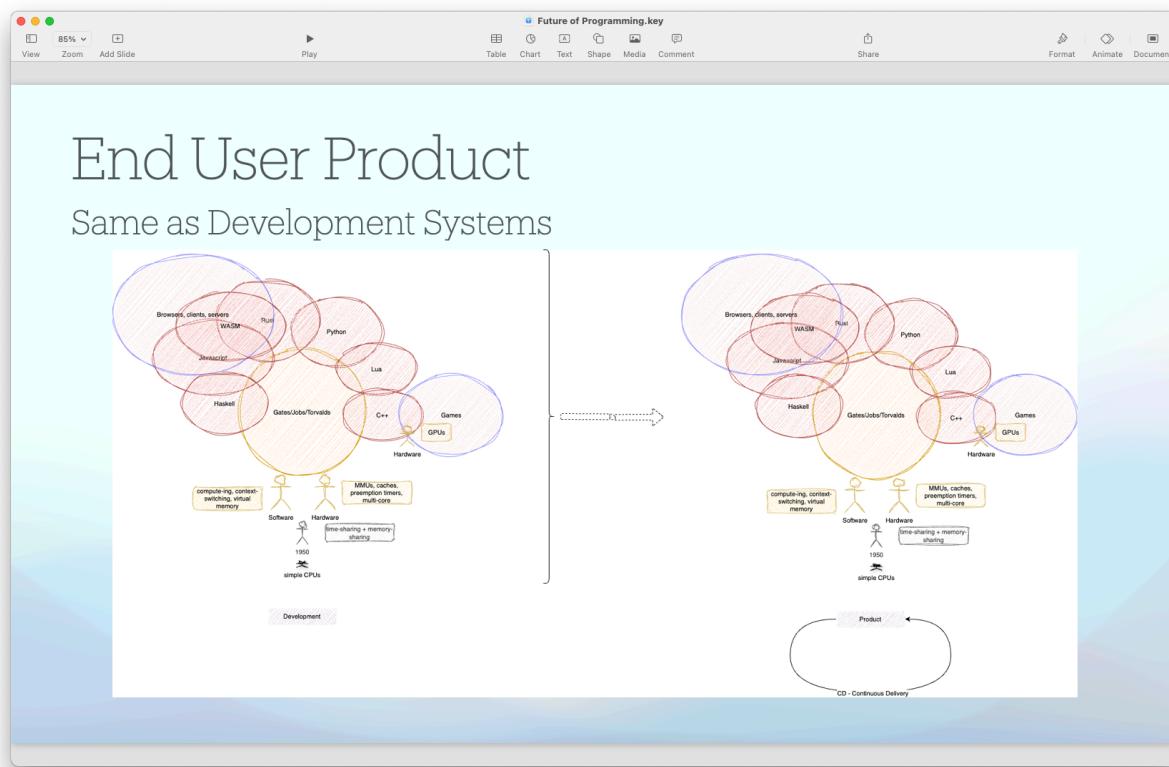
simple CPUs

MMUs, caches, preemption timers, multi-core

time-sharing + memory-sharing

deprecate this

End User Product



Idealized Productization

Future of Programming.key — Edited

View Zoom Add Slide Play Table Chart Text Shape Media Comment Share Format Animate Document

Idealized Productization

- Use Existing Software Stack for Development
- Optimize Software Stack Away Before Shipping Product

The diagram illustrates the Idealized Productization process. It shows a large cluster of programming languages (WASM, Rust, Python, Lua, C++, C, Haskell, JavaScript, and Gates/Jobs/Torvalds) at the top, connected by a yellow circle labeled "Gates/Jobs/Torvalds". Below this, a bracket groups the languages into two main categories: "Software" (left, red circles) and "Hardware" (right, blue circles). The "Software" category includes WASM, Rust, Python, Lua, C++, C, Haskell, and JavaScript. The "Hardware" category includes Gates/Jobs/Torvalds. Below the Software category, there is a box containing "compute-ing, context-swapping, virtual memory" and icons for a person and a computer. Below the Hardware category, there is a box containing "MMUs, caches, preemption timers, multi-core" and icons for a person and a computer. A dashed arrow points from the "Software" section down to a "Development" area, which contains a person icon and the year "1950". Another dashed arrow points from the "Hardware" section down to a "Product" area, which contains a person icon and the text "simple CPUs".

Infinite Memory

The screenshot shows a presentation slide titled "Infinite Memory". The slide content is as follows:

- Text-parsing is well-understood.
- AST-building is, also, well-understood, but, only if you have an infinite amount of memory at your disposal.
- Text-parsing->AST implies the need for infinite amounts of memory.

The slide has a light blue gradient background. The top bar of the application window includes standard Mac OS X controls (red, yellow, green buttons), a zoom slider set to 85%, and menu items like View, Zoom, Add Slide, Play, Table, Chart, Text, Shape, Media, Comment, Share, Format, Animate, and Document. The title bar shows the file name "Future of Programming.key".

What Is The Solution?

The screenshot shows a presentation slide titled "What Is The Solution?" in a Keynote-like application. The slide has a light blue gradient background. At the top, there is a toolbar with various icons for View, Zoom, Add Slide, Play, Table, Chart, Text, Shape, Media, Comment, Share, Format, Animate, and Document. The main content area contains the following bullet points:

- "The Solution" is not to simply add more memory or to use memory more efficiently,
- Find a dimension in which the problem isn't even a problem
- Realization of the Handmade Manifesto

Towers on Top of Towers

The screenshot shows a presentation slide titled "Towers on Top of Towers". The slide content is as follows:

- Can we solve this kind of problem by using a version of a tool
 - that wasn't sitting on top of a creaking tower of tools sold to us by Gates/Jobs/Torvalds
 - who built their tools on top of a creaking tower of tools sold to them by Intel (NS, Motorola, IBM, etc.)
 - built on top of a creaking tower of tools sold to them by wild-eyed mathematicians
 - who insisted on using up a very expensive piece of memory real-estate as The Stack Pointer that deep-down implies "blocking" (anti-asynchronosity)
 - built on top of a tower of creaking tools sold to them by IBM/Burroughs/etc
 - who didn't originally build an SP into the side-effects of other opcodes

Stream Parsing - Part of the Solution?

The screenshot shows a presentation slide titled "Stream Parsing - Part of The Solution?". The slide content is as follows:

- Stream-parsing has the advantage that it doesn't need infinite amounts of memory.
- It has the disadvantage that it doesn't give you a way to build an infinitely-sized-AST which you can crawl around in or backtrack in
- Providing infinite amounts of memory - at the expense of 55,000,000 LOC and stressed-out hardware (MMUs, caches, etc.) - is, provably, one solution.
- Is there another solution(s)?
- I've witnessed whole compilers built on streaming fundamentals (e.g. PT Pascal, Concurrent Euclid, both using S/SL (Syntax/Semantic Language)) in small amounts of memory.

S/SL[1], PT Pascal[2]

Goodbye to General Purpose Programming Languages

The screenshot shows a presentation slide titled "Goodbye to General Purpose Programming Languages". The slide content includes a bulleted list of points about the future of programming:

- The concept of “general purpose programming language” is dissolving before our very eyes
- Increased use of GPUs, FPGAs, etc.
- We need true asynchronous choreographic programming instead of more-of-the-same
- We don’t need more sticky-raisin-bun programming languages picking up dirt as they roll down a hill
 - With even more computing power and syntaxes glommed onto their peripheries

New Development Goals

Structured Message Passing in Multi-CPU Products

- Simulate multi-cpu systems
- Deprecate Time-Sharing
- Deprecate Memory-Sharing
- Product Engineering

The diagram illustrates the 'Future of Programming' through a conceptual model. It features a central 'Production Engineering' circle connected to various components. On the left, several overlapping circles represent different programming paradigms and technologies: 'Browser, clients, servers' (blue), 'WASM' (yellow), 'Rust' (orange), 'Python' (red), 'Lua' (green), 'Haskell' (purple), and 'Gates/jobs/Tornadoes' (pink). Below these are boxes for 'Architecture' (containing 'compiling, cores, switching, virtual memory'), 'Development' (containing 'Software, Hardware, simple CPUs'), and 'Product' (containing 'MMUs, caches, preemption times, multi-core'). A bracket on the right groups 'Production Engineering' and the 'Product' layer, indicating their relationship.

Bibliography

- [1] S/SL (Syntax Semantic Language) from <https://research.cs.queensu.ca/home/cordy/pub/downloads/ssl/>
- [2] PT Pascal from <https://research.cs.queensu.ca/home/cordy/pub/downloads/ssl/>

See Also

Same set of slides on YouTube:

Towers on Top of Towers



- Can we solve this kind of problem by using a version of a tool
 - that wasn't sitting on top of a creaking tower of tools sold to us by Gates/Jobs/Torvalds
 - who built their tools on top of a creaking tower of tools sold to them by Intel (NS, Motorola, IBM, etc.)
 - built on top of a creaking tower of tools sold to them by wild-eyed mathematicians
 - who insisted on using up a very expensive piece of memory real-estate as The Stack Pointer that deep-down implies "blocking" (anti-asynchronosity)
 - built on top of a tower of creaking tools sold to them by IBM/Burroughs/etc
 - who didn't originally build an SP into the side-effects of other opcodes

References: <https://guitarvydas.github.io/2024/01/06/References.html>

Blog: guitarvydas.github.io

Videos: <https://www.youtube.com/@programmingsimplicity2980>

Discord: <https://discord.gg/65YZUh6Jpq>

Leanpub: [WIP] <https://leanpub.com/u/paul-tarvydas>

Gumroad: tarvydas.gumroad.com

Twitter: @paul_tarvydas