

## **0.1. #unify\_programmers\_world**

Redefining and leading opens source by taking care of the programmer and distribution.

Imagine you have 100 devs split into 20 groups having 5 people working on a task. You'll have power. **Now imagine all 100 people walking in the same direction yielding 20x the power**

### **1. #time\_is\_money**

This is the main driving force behind this: **TIME = MONEY**

Thus minimize time allowing to reuse programmers and previous investments. **FINALLY WRITING UNIVERSAL CODE WITH LEAST COMPROMISES ?**

Desire to have it all :-)

#### **1.1. target audience & time frame**

300 million devs today. AI tomorrow. Such processes making progress and waiting for the world to gradually switch and adapt are typically 10+ years. Java was advertised with like 100 million USD in the past and turned into industry standard still talked about today.

#### **1.2. THE PROBLEM**

There are at least 2 problems to be solved

- programming languages
- software distribution (separate pitch)

Today hirings look like this: **Looking for a programmer which knows 4+ technologies It costs a lot of energy and time to stay up to date on 4+ technologies**

#### **1.3. programming languages: THE PAIN**

There are many programming languages. But none focuses on best compromise for **ALL** scenarios including backend frontend embedded speed or dynamic. **You often have to switch the horse**

They all mature and converge in some ways or another. But legacy support often prevents true progress

#### **1.4. programming languages: THE SOLUTION**

Provide a new programming language which allows all important flavors allowing to opt-in. Let user choose. The more feature, the more ideology which is good for some and bad for other tasks.

Also see #target\_environments Section 1.4.1.

You only have to learn the flowers if you need the last 5% of speed This means

- #gradually\_opt\_in
- #macros Section 1.5.12 Something lisp homoiconicity or macros thus code generating code with jitting
- #hot\_reloadable Section 1.5.2
- TS like #type\_narrowing Section 1.5.13
- #integrating\_foreign\_languages Section 1.5.7
- #code\_splitting Section 1.5.14
- which can be compiled with different focus (execution speed, size, code splitting, fastest startup time etc)
- which can be packaged or hot reloaded and differentially updated
- #blending Section 1.34.1 into the target language

- `#typelevel_function` Section 1.5.10
- REPL so that you can replace shells.
- instead of being dynamic be hot replacable ? Cause type checking is what helps and many features eg optimization require at least some analysis and known types for a section of code such as function see Julia.
- `#anonymous_types` Section 1.5.15
- optional `#opt_in_GC` Section 1.5.16
- optional `#opt_in_arena_allocation` Section 1.5.17
- maybe `#maybe_dynamic_feature` Section 1.5.3 (=open\_world)
- `#versioned_imports` Section 1.5.4
- `#typingEscapingTheCompiler` Section 1.5.5
- `#async_and_coroutines` Section 1.5.6
- `#digital_twin` Section 1.4.2
- `#proof_logic_engine` Section 1.5

#### **1.4.1. #target\_environments**

target environments and setup

- shell like dynamic usage, also See `#aot_closed_world` Section 1.11
- `#aot_closed_world` Section 1.11

The different targets might have different modes.

- optimized for runtime speed or size
- optimized for runtime speed
- with different GCs
- hot replicable (`function_bodies`, `function_bodies_and_structs`) random note that people experience it as pain having to restart on struct changes: [https://www.reddit.com/r/learnmachinelearning/comments/1bdozjp/julia\\_vs\\_python/](https://www.reddit.com/r/learnmachinelearning/comments/1bdozjp/julia_vs_python/)

Hot updatable or not which might influence some % of the performance.

- `#targeting_JavaScript` Section 1.5.8
- `#targeting_WebAssembly` Section 1.5.11
- `#targeting_native` Section 1.5.9
- with `#opt_in_GC` Section 1.5.16 cause less programming work. Its often ok to trade 5% of speed for being many times more productive.
- low latency mode ZGC like garbage collection no GC (rust like)
- partially loaded (web)
- orchestrated and updatable (Elixir like) mobile like eventually with dev support running on PC which sends updates jitted code and upgrades By this I mean cross device platform orchestration pushing changes and updates to multiple devices hot replacing or restarting code.
- arena allocations / linear memory for special use cases
  - ▶ web servers
  - ▶ GPU graphics and gaming

#### **1.4.2. Why #digital\_twin**

Eg ios cannot jit. But you can send new code as shared library. Same pattern exists eventually for embedded or mobile if you want to send shaders or julia like. Eg Taichi. So the source code (one twin) generates the binaries (executable, second twin) which then runs on the device.

#hot\_reloadable Section 1.5.2 ...

## 1.5. Why #proof\_logic\_engine

Eg see Example Julia: Must carefully be ..

If you allow compile time arguments eg proofs you can avoid checks sometimes. Examples

```
div = (a, b) => a / b
head = (A, B) => A[B]
```

Now if you can proof that B always in A or b always zero you can avoid the check. **BUT** it will lead to duplicate code. Thus will make most sense for inlined functions.

### 1.5.1. #gradually\_opt\_in

Most languages suffer from idealism. They think they know what's right. But honestly it always depends on the project. So the idea is to allow moving targets with opting in. And opting in to many features yields idealism. But you have the choice and don't have to switch tools Yes, this pulls complexity into the language. But less than using the many languages eventually cause simple things will be simple in all

### 1.5.2. #hot\_reloadable

Mun-lang, ErlangVM. Today you have 2 choices:

- hot reloading (requires much testing, as much downtime as rewriting the memory eventually requires or rewrite on access)
- rolling replacement (micro architecture, eventually no downtime)

So hot code replacement is most useful for

- GPU, OpenXR, Gaming
- hot patching (eg linux kernel without taking system down)

Picking the sweet spot for development maybe for production if people want:

- (Rust, Java, Nim, Kernel) do hot replacement of function bodies
  - Gaming engine allow to hot reload resources
  - JS eg Vite allows to hot replace modules and turned into standard
  - upgrade structs and data types (This might not make sense for production servers cause rewriting huge memory takes time) but for eg game or GPU development for code which manages state
- Mun shows how this is done

### 1.5.3. Why = #maybe\_dynamic\_feature

JS, Python, Ruby, Julia, Nix, PHP are dynamic in the sense that you can pass any type everywhere without worrying. However once you want working code you want type checking. And then #hot\_reloadable Section 1.5.2 is enough? But if there is a real use case dynamic support will be added

Dynamic has the following issues:

- when changing code (eg structs) you want to patch all instances and the code creating them. Dynamic will not do this
- Eventually you want type checking cause runtime errors maybe later is always worse

The only real use case for dynamic I see is laziness: "Let me intentionally break code and focus on a small details if that works let me finish the refactoring"

So you can reload one module and fix the others later.

#### **1.5.4. Why = #versioned\_imports**

If you have hot reloading or if you want to transform older versions of data written by software you have transform the previous and the new data types.

If you allow import module\_version\_hash you can get both done in a nice way by importing current and previous version and writing transformations.

This might be a game change for >hot patching< production systems such as linux kernels **WHILE** keeping state. Imagine you fork and fix and your kernels update without loosing a beat **WHILE** you keeping control over what was patched cause the kernel is the digital twin of the fork.

#### **1.5.5. Why #typing\_escaping\_the\_compiler**

```
row = sql: from users
println(row.phone as string[20])
```

If phone doesn't exist yet allow the dev system to write a migration file adding the column.

#### **1.5.6. #async\_and\_coroutines**

You can't do without anymore I think (Rust's tokio, JS Promise, Python's asyncio & trio).

The problem is more than the systems are different. Like trio or taskgroups allow to spawn and cancel everything.

JS/Rust don't allow this you have to manually pass cancellation tokens ?

Where is #gradually\_opt\_in Section 1.5.1

#### **1.5.7. Why #integrating\_foreign\_languages**

- Example Numpy: Python gets used by embedding Python due to its array syntax
- execution contexts: Calling into server or database, or GPU commands
- calling into database (SQL)
- ansible, SASS do exactly this: They mix a language and a target DSL
- command line arguments can be seen as Array language with own completion and rules for each executable

#### **1.5.8. Why #targeting\_JavaScript**

JS because its fastest for browsers and can be split

#### **1.5.9. Why #targeting\_native**

its 1.5 - 3x faster than Webassembly

#### **1.5.10. Why #typelevel\_function**

Targeting native and JS at the same time eventually leaves some choices. structs could be represented as memory block (eg when targeting Vulkan from JS), as object or even as array replacing keys by index access. When targeting native could be struct or hash (mimicing objects which are extensible).

And type level functions allow to determine the target type depending on the target language.

#### **1.5.11. Why #targeting\_Webassembly**

- sandox used by browser competing with JS

#### **1.5.12. Why #macros**

- Because you want to send generated code to client, not your database scheme

### **1.5.13. Why #type\_narrowing**

- TS simple example: fetch('') as Promise<{temperature: number}> Is much easier than what you can do in traditional old languages including Delphi, Java, C++. You either have to recreate the type narrowing by creating classes or you're untyped. Both are worse solutions
- Haskell: You can't just say from int use 0..9 you have to create your own Digit type and converters (what for?)

See #cooperating\_with\_existing\_languages

### **1.5.14. Why #code\_splitting**

- webassembly
- JS

To have fast websites. Long loading time = loosing sales > 2 secs eveny additional sec 10% less some studies tell

### **1.5.15. Why #anonymous\_types**

row = sql: ... Would you love having to define the row types if it can be derived from the database ?

### **1.5.16. Why #opt\_in\_GC**

Avoiding GC means having to take care about an owner or use RC and cycles. So its often the best compromise to get your job done. Go/Swift/PHP/JS/SH/Python/Nim shows it can be fast enough for most cases but means zero overhead when typing

### **1.5.17. Why #opt\_in\_arena\_allocation**

Best for gaming and graphics cause you need stability for each

## **1.6. #cooperating\_with\_existing\_languages**

This is later stage. Anyway: Most languages have a very compelling selling point. Java: Weaving, OOP, cross platform. Nim: Multiple GCs choosable Zig: Simple, pass GC to be used. TS: Type narrowing and strong type level function support

And some features such as TS's type narrowing and strong type level support makes sense for other languages, too.

Because sometimes rewriting isn't an option, supporting and improving is. Another example is having cross platform APIs:

\$s = '() => 2' as String<IfEvaluated<TS: () => number>> can't be done but would be useful. RakasJS useServerSideQuery allows to use the typing of datatypes of server code to define the typing of a REST API on the fly. This could be used for mutliple languages but typing solutions can't follow yet.

Currently targeting nativen (C like APIs) and JS is planned. Maybe more languages might follow See #blending Section 1.34.1.

### **1.7. #examples\_for\_optional\_traits**

- borrow checking
- different gcs
- 32bit pointers on 64 bit systems (like JS engines use)
- ahead of time compilation, multiple ways to jit, hot replacement, optimizaiton (size/speed)
- way of distribution
- for C eg ABI

## 1.8. Why is unifying vs splitting the programmers world a big deal ?

Examples:

- julia not good webassembly support
- Rust OpenXR doesn't work on Pico4/Quest2
- Go no OpenXR bindings yet
- Nim -> hot reloading didn't work
- Vim XML library -> had to patch it back then
- jackd was written in C and C++ cause programmers couldn't settle which one is better
- history of QT: C++ + Python + QuikQT (JS layer) So you get 70 MB distributions and huge mess cause you have to learn things 2-3 times
- npms' node-gyp can compile C++ but not Rust yet
- imgui (C++) vs imgui-python (some features missing or they don't work you only find out after jumping in)

By unifying the programming standards you remove the source of the evil.

## 1.9. putting the creator (ai/programmer) first

There was a study that programmers roughly write the same amount of code no matter what language. Maybe because writing typing (fater thinking is done) is the bottle neck. So the more powerful the language the more powerufl the user.

Why does it also help AI? The higher quality the structure the better the result.

Proofing the best compromise for combination langugae, editor, programmer has been found by running competitions hoping the owns solution is used yielding top results.

This will give **THE TRUST WE NEED TO SUCCEED**

## 1.10. Goals language

- harmonizing fast vs slow compilation, hot reloading, ahead of time compilation, differential updates interpreted mode and orchestration on native and Webassembly maybe JS
- Eg Vite/Micropython prepackage libraries while being more dynamic on the code you work on.  
-> story #targeting\_esp32
- Lisp like macro system and code rewriting which **then generates code** which **then can be type checked** ahead of time or at runtime. This doesn't exist yet Use case 1: Dual use of code backend frontend (RakkasJS/Qwik City/ .. like frameworks eg useServerSideQuery rewriting etc)
  - provide good completion like TS which **can escape** the compiler and write database migrations for you
  - harmonizing Rust vs JS vs Java Eg Vscode vs Eclipse vs Lapce (...) If you want to provide plugins you have to provide the many today
  - harmonizing even executable vs library ? Why can't a library provide its own completion and why can't you call it from a shell ? What's the difference ? (ok cleanup - but still) So why do programmers have to learn the many ways how to use git ?

So make shared libraries provide a functions for execution with args and and complete shell and you no longer need executables cause the shell can spawn a helper loading the shared library

## 1.11. #aot\_closed\_world

AOT (closed world) makes sense for mobile devices and is a requirement for ios which doesn't allow jitting and because interpreted is slower aot is the only option left. The closed world means you know all the code which might be running ahead of time.

AOT isn't as bad as it sounds because you can #hot\_reloadable Section 1.5.2. And this good enough to build a repl, eventually.

And if this is not good enough #maybe\_dynamic\_feature Section 1.5.3 could be implemented.

And the open question is can we implement a partially closed world. Like a piece of code to be run getting optimized.

```
id
# the fields to be fetched can only be known if the row turns closed.
# otherwise just all could be fetched
row = sql: FROM users WHERE id= $ID
println(row.name)
close(row) # now println can take place :-P
```

So the interpreted version of closed worlds leads to lazy code waiting for things to be closed before they can execute.

But this is complicated much overhead so should be later experiment

## 1.12. Why other languages fail

Failing means they **fail to satisfy all needs of a programmer**. They are good for what they have been written for - no doubt. While goals sometimes are orthogonal to each other the open question remains is there a way to build better programmer experience and be more flexible with less compromises.

### 1.12.1. Why idealism fails (for all possible use cases)

All languages which force one direction, the loose in another.

### 1.12.2. Why Lisp fail

I tried sbcl and even a simple math function was hard to optimize and get up to speed. AND a.b.c completion on struct or object like data is time saving. Lisp doesn't want to and doesn't provide. You might work around it by introducing your own syntax

Lisp shows the power of macro programming

### 1.12.3. Why Lua fails

See <https://mun-lang.org/> doesn't perform well on all target platforms

### 1.12.4. Why Mun-lang fails

<https://docs.mun-lang.org/ch04-04-hot-reloading/> take from it:

- hot reloading

good:

- Has hot reloading built in !
- targets webassembly and native

bad:

- no updates since Mai/June. Maybe because Rust has hot reloading. But that cannot change structs eg add a field So the main issue having perfect dev experience isn't reached. And what's best for production well let the user choose.

- depends on old LLVM again a distribution problem nixos no longer ships that old one.  
#fix\_distribution\_problem
- No code splitting for web ?
- does it only have devs in mind (for now) - could be lifted
- you to use the GC and runtime. Can you create a Rust like high performance build later? you have to stick with registering function calls multiple times. You'll never get rid of the runtime if you want to. Maybe its bearable.

### **1.13. Why PHP fails**

Well 40% are Wordpress, so. But its only server side. You can't write unified code (you can but then you get 20 MB WebAssembly download so your site would be slow)

PHP showed that if you serve HTML you get somewhat easy maintainable and eventually fastest code. JS didn't reach this kind of web programming until Qwik City was born.

#### **1.13.1. Why JS fails**

Well JS/TS actually does very well.

- TS: no macros (bun is an exception) This is required to generate code and then have the typing engine type it. Eg deriving a form from database schema without exposing the schema to unsafe client side
- not for big data eg C/Rust like 5-10x faster
- Vite like frameworks require dependencies on Go and Rust and JS and 2 different bundling mechanisms

JS has a good eco system today.

#### **1.13.2. Why TypeScript fails**

See JS. Its run by Microsoft and they made a statement. Will never support macros. (bun escaped .. but ..). So you'll never be able to generate code from a schema without having to send the schema (security issue!) to the client.

#### **1.13.3. Why Rust fails**

good

- one package manager (cargo) streamlining the community
- backward compatibility

bad

- slower to write than JS
- some patterns require unsafe
- You don't have hot upgrades like elixir
- You don't have a choice. You must implement low level future checking
- they keep anonymous records. But its a somewhat not well documented feature. But its requirement to derive types from SQL queries

#### **1.13.4. Why Elixir / Erlang VM fail**

good you can patch every module and write servers you never have to take down

bad maybe performance not widely used outside of that use case

#### **1.13.5. Does vite fail?**

Imagine you want to manage 500 similar pages. If you use Vite you'll end up with 500 repositories having to manage write and read the many files. Why not just do everything in memory and be done ?

```
apps = map create_app (1..500) run(apps)
```

So Vite doesn't fail for one project. But once you want to manage the many it starts turning time consuming and awkward

### **1.13.6. Why does Python fail**

Well, its one of the top most used languages. But targeting AMD GPU requires knowin what packages to choose. Its not the fastest. That's why proprietary solutions have been created. Sometimes packaging doesn't work. While you can optimize some functions its hard to optimize bigger chunks

### **1.13.7. Why Go (partially) fails**

Its a very good compromise and does its job very well. Eg if you want to target OpenXR (VR glasses) the work hasn't been done yet.

### **1.13.8. Why julia fails**

good

- homomorphic like lisp
- dynamic (only optimizes what gets used)

bad:

- embedded android and others require AOT. Julia was written without AOT in mind. So while they got there eventually its constant fear of something breaking.
- the runtime requires LLVM compiler which is hard to package for mobile and the web. There are tools which compile ahead of time but they are still somewhat experimental So it was written for strong machines and everything else seems to be second class citizen
- there is no real support in LSP while developing kernels You have to run to get typing issues
- not enough static guarantees. Nice for playing around .. but in the end you want to catch the errors before the customers. You have no clue whether a kernel will run before you actually do so.

### **1.13.9. Why Agda Urweb and some more academic languages fail?**

The are very ideologic. Prooving that 2 lists in zip are equally long is fun. But what if you have streaming data from files ?

Eventually take from Urweb: Passing proofs to functions.  $(a, b) \Rightarrow a/b$  if you can proof that b never is 0 no need to check for the condition. -> #proof\_logic\_engine Section 1.5

Urweb: too complex to narrow in scope. But many nice ideas

### **1.13.10. Why C# like fail**

- If you run on embedded jitting and launch times are incredible long
- well not open source

### **1.13.11. Why Zig fails**

- You don't have the Shell/ruby/python like careless usage you have to pass GC

Would be easy to fork and fix eventually.

### **1.13.12. Why Nim fails**

bad

- The author thinks its a feature having to annotate simple code with a return type. The truth is it takes time.

good

- Its a nice language with (too) small community which finds a nice compromise and shows that its easily possible to implement the many different GC versions

### **1.13.13. Why Haskell fails**

good:

- typing system

bad:

- speed. People sometimes use for prototyping then rewrite in Rust. Thus twice the work

### **1.13.14. Why HaXe fails**

bad:

- it isn't true abstraction. While it targets the many PHP, C, Java the details matter and its hard or impossible to truly write cross platform code. You have to adopt the language usage to the target

It's still a very nice project and community no doubt.

### **1.13.15. Why Ocaml fails**

Well cool language no doubt. You have to opt-in for scientific simd like optimization.

While there are forks for hot loading, allowing SIMD like or llvm abstractions I bet you hit a lot of incompatibilites and walls switching between the solutions

OxCaml ? <https://oxcaml.org/> -> fearless concurrency -> layout & extensions -> tools to control allocation -> only x86-64 and aarch64 ? Support for NEON and AVX512 is coming soon. <https://github.com/let-def/hotcaml>

### **1.13.16. Why Dart fails**

Its OO only same as Java. Yet it works for apps etc hot reloading keeping state is done well

### **1.13.17. Why Java fails**

bad:

- you can't optimize objects well
- they are about maybe eventually somewhat introducing vector optimizations and similar others already have ?
- OO doesn't fit all use cases
- HotSwap only allows changing code, but not adding fields It's ZGC GC shows a path how it might be possible to change layout of memory.

good:

- stable
- industry standard

## **1.14. Details why learning the many languages is that expensive?**

Learning a language

- syntax
- core language module
- language server
- package manager and how to express dependencies conditional compilation and more
- extra language models
- hot reloading
- The libraries how to get a job done

- **DSLs** eg how to write SQL using objects ?
- web / async

## 1.15. The consequences

Smart people want to tell you if you know one language you learn the other faster. But reality is because languages are powerful learning the many details doesn't really make you faster cause each function is its own beast. MIT studies (forgot) told that in IT knowledge transfer isn't possible. Well AI tried translating C API into JS but the JS one didn't exist **lol**. Its not just humans

## 1.16. the consequences

People reinvent the wheel

- Gimp vs Krita
- vscode vs lapce
- .net vs .. whatever

Eg opencv being C++ having Python bindings and its a mess cause you never know which documentation or code works r is current. => **2x the work**

Vite, webpack & Rust -> **3x the work!**

## 1.17. shortcomings and bottle necks

TODO move into the why fails section

- Haskell -> too slow sometimes
- JS/TS/PHP/Ruby/Python -> too slow
- Go: very good, but worse at targeting JS and no OpenXR bindings yet
- Rust -> gets many things done but some patterns just don't fit Rust
- Zig/Nim/Rust: Always requires all type annotations (taking time)
- Zig will not have Rust like iterators (cause its an imperative languages)
- Zig wants to pass an allocator always
- TS will never have macros (but implemented them) so you're stuck cause macro support must be within the compiler its not packaging problem
- Haxe: Will never have short lambdas
- PHP: Server side only. Java/.net was tried failed nobody picked it up
- Rust: Skia like graphic library officially stating **we are not as fast as the C version yet.**

**WHO WANTS TO FOCUS ON THE PRODUCTIVITY OF THE DEVELOPER** rather than trying to follow a **narrow goal** which should be **OPTIONAL**.

## 1.18. Why pressing SQL into objcets isn't ideal (eg POJOs)

Many lanugages abstract SQL by using object syntax. But how do you optimize by letting compiler figure out you never need field X and not even loading it ? You would need partial classes whatever So you'll never be able to write most efficient code with OO abstractions.

## 1.19. what we really need

A compiler which is esperanto (arrays,hashes,classes) which can be compiled down to native for fastest backend, but also to lightweight JS allowing to gradually switch from JS to Rust like features depending on project needs without starting from scratch.

supporting

- compiled properties (which can be used in an jitted or interpreted way)
- with a JS like dynamic layer on top cause some problems are dynamic
- embedding other languages.

- multi language aware typing systems so that PHP types can be converted to JS types etc
- targeting UIs:
- GPU (imgui taichi like)
  - HMTL
  - native UI librarise (React like?)

## **1.20. why this unifies the dev community ?**

Cause it unifies many splitting points

- shell vs language ? If shells can be completed within higher level language which can be interpreted no more need for a shell ?
- if no more 10 languages are required the community is 10x stronger !
- ..

## **1.21. the biggest challenge**

It will require time. Legacy and wisdom in brains takes time to change. So growing slowly is best achievable solution

## **1.22. insanity example Qt**

- historically C++
- historically C++ sucked, so added Python layer
- Want to be modern (QwickQT) so adding JS properties and dynamic ..

But in the end you just need GPU/ HTML targets today maybe native look and feel

## **1.23. random pain points**

YOu can't target JS in PSQL (plv8 = no async) and querying PSQL async with the same code easily.  
**2x the work**

## **1.24. embedding examples**

- SQL within any language
- calling into server (Rest API) see useServerSideQuery examples
- managing execution contexts
  - other browser plugin context
  - other threading context
  - other JS execution context
  - JS Webassembly
- command line tools within languages
- Rust-GPU/tachi/.. others They try to abuse the host language to generate GPU code. compting with shader-lang etc

## **1.25. the potential impact: Examples**

Job description: **Use Ruby, need fast running code** Reality: **Switching from Ruby to Rust means removing 4 from 5 servers cause one is enough**

Problem is if you start with JS you can't switch. You have to rewrite.

If you start with Rust you can't switch

## **1.26. #targeting\_esp32**

If you use Micropython you get 1sec restart times to test your code. If you use C++ you get 25 seconds (compilation + transfer) update times. Why ? Cause you have to flush all the app. If you had differential updates C++ like udates would be equally fast

## **1.27. special case: targeting JS**

By allowing to define datatype functions which turn into a datatype once the target is known  
structs objects etc can be handled for both native and JS.

## **1.28. special case: targeting embedded**

Embedded devices have slow CPU and not many resources. This means all the compilation must be done outside of the device. Thus the dev env must send the and push the updates and the target device eventualaly should cache them so that its self contained and survives a restart

## **1.29. special case: shell scripting**

Because its interpreted or compiled with fasts but little bit slower jitting it will be able to be used as shell scripting language. Because we treat command line arguments as DSL with its own LSP command line completion etc will be possible.

## **1.30. special case: webassembly**

WasmGC or custom GC or no GC. We'll support all Jitting we'll support by either asking the dev env to push changes or by embedding a small jitting engine which builds wasm modules and loads them.

## **1.31. special case: web server backend**

by using arena memory like impredicative.com ur we'll have best possible reply times.

## **1.32. special case: resumable components, react like and others**

We will translate Qwik City logic. Our advantage will be that the backend might be little faster and have more throughput, see #js\_still\_being\_slow Section 1.35.1

## **1.33. disrupting and replacin**

Go, JS, Rust, provisioning systems, vscode, Python, Zig, Github, .. GItub review, code hosting, compilation .. is side story

## **1.34. glossary**

### **1.34.1. #blending**

This means if you target JS you can write JS and not JS pressed into the original language. It also eventually means when targeting totally different systems to limit the core language to a common denominator with target specific extensions

## **1.35. references**

### **1.35.1. #js\_still\_being\_slow**

<https://sancho.dev/blog/server-side-rendering-react-in-ocaml>

TODO there is so much missing here

## **1.36. Example why too much idealism hurts some cases**

People want to break out, always.

So its time to see language features as traits, not a reason to learn a new eco system ..

- Rust -> the rhai got a competitor with GC cause its simple for scriping
- Rust -> Alloy fork introducing GC<> type but not available on OSX yet (2025-12-22)
- SASS with embedded scripting languages
- ansible noticed >Hey, we need loops<
- Taichi compiling for the traged GPU (jitting for GPU)

- JS having many heavily optimzied features including 3step jitting
- LISP is cool except its not (performance wise) - Thus a niche

### **1.37. How to make people use it and help the switch?**

- by also providing eduation platforms

### **1.38. How to make it easy for people to adopt ?**

- for julia users julia names ?
- for Python users Ppython names ?
- ....

### **1.39. How to make money ?**

Well this will touch

- simulation
- ai
- embedded
- Distribution as software apps and web native Webassembly JS

So will have multiple markets and ways to make money

Well. You can't make money by open source. If you are not open source adoption is slower. The plan is to build multiple services and killer apps

- Github like + compilation and distribution servers
- Unrealengine & Unity like but with proper hot code reloading ?
- subscription based open source solutions (we don't eat your data but you pay us 1 USD/ year but you can tell us what you hate and we'll fix)
- (micro) education and upwork like market places etczz
- future AI serverices: We train on your code so that AI knows your code and makes you more productive
- selling books services about new features and best practises
- trainings
- support
- Julia got many sponsors from science. This concept competes with Julia **AND** with other embeddes solutions. So maybe there will be even more sponsors <https://news.ycombinator.com/item?id=27883047>
- provide dev envs, notebooks etc which can be 'ordered' by schools etc
- maybe even space industry cause hot patching in space the easier faster the better
- compete with shopware cause we'll have better and faster technology