Designing a Programming Language for the Desert

Troels Henriksen

DIKU
University of Copenhagen

2022-02-06

Background

Me: Troels Henriksen, researcher at the University of Copenhagen.

Team: Cosmin Oancea, Philip Munksgaard, Robert Schenck, Martin Elsman, Fritz Henglein, former and future students, Internet people...

Project: **Futhark**, a purely functional parallel array language.

Futhark, briefly

- Fast, flexible ML-like language for high-performance computing.
- Compiles to parallel GPU or CPU code.
- **Aggressively optimising compiler** (this is what we publish papers about).

```
def dotprod [n] (a: [n]f32) (b: [n]f32)
  : f32 =
  reduce (+) 0 (map2 (*) a b)

def matmul [n][m][p] (a: [n][m]f32) (b: [m][p]f32)
  : [n][p]f32 =
  map (\a_row -> map (dotprod a_row) (transpose b)) a
```

- **Not intended for full applications**, only the small performance-critical parts.
- This talk is not about the language or compiler itself, but **general principles** we've used for designing an obscure language.

Building a programming language takes hubris

- **The average user count over all programming languages is close to zero.**
  - Language designers know this.
  - Obviously their ambitions go beyond this.

Building a programming language takes hubris

- **The average user count over all programming languages is close to zero.**
  - Language designers know this.
  - Obviously their ambitions go beyond this.

Most languages are designed with the hope of great success!

- General-purpose or with a very large domain.
- Must scale to large teams, large programs.
  - Will have and need complex build tools, debuggers, package managers, etc.
  - Might even have one of those *sufficiently smart compilers!*
- Most users will have the language as their main language.
  - Time and motivation to learn many details.
- Meant for a **resource-rich environment**.
  - Not about machine resources!

Companies think like this when pushing a new language, but hobbyists often do too.

Some programming languages built for success

Rust	Cone	C3	Inko	Crystal	MANOOL
Myrddin	**Go**	Raku	**Java**	Mercury	**Swift**
SML	OCaml	**PHP**	Clean	Racket	Erlang
ReasonML	Smalltalk	Groovy	D	Dart	Oberon
C#	Plasma	Zig	PureScript	**JavaScript**	Haskell
Julia	R	F#	Clojure	Ruby	Scala
Eiffel	Nim	Elixir	Odin	Kotlin	Solidity
and so on...					

- **Bold ones** may now have enough resources for “sufficient tooling” to exist.
  - Some always had due to corporate support (Swift).
  - Others because they became popular organically (Rust).

So most languages are intended to be this

https://commons.wikimedia.org/wiki/File:Bengal_tiger_(Panthera_tigris_tigris)_female_3_crop.jpg

Bengal tiger

What about Futhark?

Domain: High-performance parallel number crunching.

Users: Typically programmers who mostly use some *other* language and want to speed up some part of their program.

Usage: Will be a *guest* in a larger code-base not written in Futhark.

What about Futhark?

Domain: High-performance parallel number crunching.

Users: Typically programmers who mostly use some *other* language and want to speed up some part of their program.

Usage: Will be a *guest* in a larger code-base not written in Futhark.

This is not a resource-rich environment!

Even in the (improbable!) *best case* of total dominance in its domain, Futhark will never have many users or many resources behind its development.

So this is Futhark

https://commons.wikimedia.org/wiki/File:Desert_Hedgehog.png

Desert hedgehog

Language design for the desert

Our approach is a kind of conceptual minimalism.

- Minimize things that require ongoing maintenance.
- Minimize implicit behaviour.
- Minimize degrees of freedom.
- Minimize novelty.
- Do just a few things, so that you can do them well.
- **Say *no* to things that are good ideas in most languages.**

Language design for the desert

Our approach is a kind of conceptual minimalism.

- Minimize things that require ongoing maintenance.
- Minimize implicit behaviour.
- Minimize degrees of freedom.
- Minimize novelty.
- Do just a few things, so that you can do them well.
- **Say *no* to things that are good ideas in most languages.**

Let's look at some concrete examples.

Build systems and multi-file programs

Nobody enjoys learning about build systems or import mechanisms.

- While Futhark is for *small* programs, we still want to support *multi-file programs*.

Principle

The easiest thing to learn is something you already know.

File imports in Futhark

```
import "foo/bar"
```

- Imports the file `foo/bar.fut` *relative to the importing file*.
- All uses of code in other files must be through explicit import.
- Pro: **Just normal filepath semantics!**
- Downside: files have no canonical name.

File imports in Futhark

```
import "foo/bar"
```

- Imports the file `foo/bar.fut` *relative to the importing file*.
- All uses of code in other files must be through explicit import.
- Pro: **Just normal filepath semantics!**
- Downside: files have no canonical name.

Example of importing the **bolded** file

```
main.fut
foo/
    bar.fut
    baz.fut
quux/
    bar.fut
    baz.fut
```

File imports in Futhark

```
import "foo/bar"
```

- Imports the file `foo/bar.fut` *relative to the importing file*.
- All uses of code in other files must be through explicit import.
- Pro: **Just normal filepath semantics!**
- Downside: files have no canonical name.

Example of importing the **bolded** file

```
main.fut ................................import "foo/bar"
foo/
    bar.fut
    baz.fut
quux/
    bar.fut
    baz.fut
```

File imports in Futhark

```
import "foo/bar"
```

- Imports the file `foo/bar.fut` *relative to the importing file*.
- All uses of code in other files must be through explicit import.
- Pro: **Just normal filepath semantics!**
- Downside: files have no canonical name.

Example of importing the **bolded** file

```
main.fut ...............................import "foo/bar"
foo/
    bar.fut
    baz.fut .................................import "bar"
quux/
    bar.fut
    baz.fut
```

File imports in Futhark

```
import "foo/bar"
```

- Imports the file `foo/bar.fut` *relative to the importing file*.
- All uses of code in other files must be through explicit import.
- Pro: **Just normal filepath semantics!**
- Downside: files have no canonical name.

Example of importing the **bolded** file

```
main.fut ..................................import "foo/bar"
foo/
    bar.fut
    baz.fut ...................................import "bar"
quux/
    bar.fut
    baz.fut ..............................import "../foo/bar"
```

Why is this the right choice for Futhark?

- *Not* textual inclusion as C's `#include`.
  - Each file must still be syntax- and type-correct by itself.
- No "search path" set by some build tool config file.
- Compilation is just `$ futhark cuda main.fut`.

Why is this the right choice for Futhark?

- *Not* textual inclusion as C's `#include`.
  - Each file must still be syntax- and type-correct by itself.
- No "search path" set by some build tool config file.
- Compilation is just `$ futhark cuda main.fut`.

Tooling advantage

- **If a Futhark program can compile as a whole, then each constituent file can also be used directly as a "compilation root" by the compiler.**
- Makes it super easy to write simple yet functional tools:
  - Emacs mode can just pass whatever file is open to the compiler to get type errors—no need to think about any build system (there is none).
  - "Go to definition" works with zero configuration, too.

Why is this the right choice for Futhark?

- *Not* textual inclusion as C's `#include`.
  - Each file must still be syntax- and type-correct by itself.
- No "search path" set by some build tool config file.
- Compilation is just `$ futhark cuda main.fut`.

Tooling advantage

- **If a Futhark program can compile as a whole, then each constituent file can also be used directly as a "compilation root" by the compiler.**
- Makes it super easy to write simple yet functional tools:
  - Emacs mode can just pass whatever file is open to the compiler to get type errors—no need to think about any build system (there is none).
  - "Go to definition" works with zero configuration, too.

- **Definitely not the right choice for every language!**
  - No notion of "shared libraries", since all paths are relative to each file.
  - Package installation must put files in a known and accessible location.

So let's talk package management

Language package managers solve tricky problems.

- How do we find packages and make them available to the compiler?
- How do we deal with conflicting version bounds in dependencies?

So let's talk package management

Language package managers solve tricky problems.

- How do we find packages and make them available to the compiler?
- How do we deal with conflicting version bounds in dependencies?

This can get really complicated.

- Central registry of packages.
  - We need a server... but desert survival doesn't leave much time for server management.
- Version bounds on dependencies, often both upper and *lower*.
  - Requires an NP-complete solver.
    - Very difficult to explain conflicts to the user in a comprehensible way!
    - Rust's solver in `cargo` is thousands of LOC.

`futhark pkg`: the simplest thing that could possibly work

`futhark pkg` is not much more than a glorified file downloader.

- Add dependency on some library to `futhark.pkg` file[1]:

```
$ futhark pkg add github.com/diku-dk/sorts
```

[1] Currently packages must be GitHub or GitLab repositories, but this is not a fundamental part of the design—we just need a way to get a list of available versions.

`futhark pkg`: the simplest thing that could possibly work

`futhark pkg` is not much more than a glorified file downloader.

- Add dependency on some library to `futhark.pkg` file[1]:

```
$ futhark pkg add github.com/diku-dk/sorts
```

- Download dependencies to `lib/` directory:

```
$ futhark pkg sync
```

[1] Currently packages must be GitHub or GitLab repositories, but this is not a fundamental part of the design—we just need a way to get a list of available versions.

The `lib/` directory after `futhark pkg sync`

```
$ tree lib
lib
└── github.com
    └── diku-dk
        ├── segmented
        │   ├── segmented.fut
        │   └── segmented_tests.fut
        └── sorts
            ├── bubble_sort.fut
            ├── bubble_sort_tests.fut
            ├── insertion_sort.fut
            ├── insertion_sort_tests.fut
            ├── merge_sort.fut
            ├── merge_sort_tests.fut
            ├── quick_sort.fut
            ├── quick_sort_test.fut
            ├── radix_sort.fut
            └── radix_sort_tests.fut
```

Package versions

- Versions are `git` tags:

```
$ git tag vX.Y.Z
$ git push --tags
```

- Packages can depend on minimum versions of other packages.
- `futhark pkg` must also downloads dependencies-of-dependencies.

Version resolution

Ross Cox from Go came up with a really simple system.

The Minimum Package Version (MPV) Algorithm

- Use the *lowest* version of a dependency that satisfies all constraints.
- Constraints on upper bounds *not possible*.
- Breaking backwards compatibility counts as an entirely distinct package
  - The SemVer major version number is part of the package “name”.

Version resolution

Ross Cox from Go came up with a really simple system.

The Minimum Package Version (MPV) Algorithm

- Use the *lowest* version of a dependency that satisfies all constraints.
- Constraints on upper bounds *not possible*.
- Breaking backwards compatibility counts as an entirely distinct package
  - The SemVer major version number is part of the package "name".

Con: Breaking compatibility in small ways or accidentally is very awkward.

Pro: Go uses it, so it is not *fatally* flawed.

Pro: Version solving is reproducible without freeze files.

Pro: Only way solving can fail is if a package does not exist.

Pro: Implementation is *extremely simple*.

The MPV algorithm in Haskell

```haskell
doSolveDeps :: PkgRevDeps -> SolveM ()
doSolveDeps (PkgRevDeps deps) = mapM_ add $ M.toList deps
  where
    add (p, (v, maybe_h)) = do
      RoughBuildList l <- get
      case M.lookup p l of
        -- Already satisfied?
        Just (cur_v, _) | v <= cur_v -> return ()
        -- No; add 'p' and its dependencies.
        _ -> do
          PkgRevDeps p_deps <- getDeps p v maybe_h
          put $ RoughBuildList $ M.insert p (v, M.keys p_deps) l
          mapM_ add $ M.toList p_deps
```

Not Futhark-specific

The `futhark pkg` design was also used for an SML package manager:

`https://github.com/diku-dk/smlpkg`

An easy-to-implement design for any minimal language (1506 LOC of SML in total).

Design details

- `https://futhark-lang.org/blog/2018-07-20-the-future-futhark-package-manager.html`
- `https://futhark-lang.org/blog/2018-08-03-the-present-futhark-package-manager.html`

Other examples

- **Use a familiar programming model:**
  - Futhark is basically a subset of “common” functional concepts: `map`, `reduce`, `scan`, higher-order functions, type inference, etc.
  - Language novelty only in very select places.
  - …but lots of novelty in the compiler itself.

Other examples

- **Use a familiar programming model:**
  - Futhark is basically a subset of “common” functional concepts: `map`, `reduce`, `scan`, higher-order functions, type inference, etc.
  - Language novelty only in very select places.
  - ...but lots of novelty in the compiler itself.
- **Support very few compiler options:**
  - Cause combinatory explosion of code paths—difficult to test.
  - *Especially* options that affect code generation or optimisation.
  - **Fun game:** see if the Linux kernel can compile correctly using randomly selected optimisation options for GCC.

Conclusions

- **Designing a programming language for the desert** means coping with *persistent* scarcity of both users and maintainers.

Conclusions

- **Designing a programming language for the desert** means coping with *persistent* scarcity of both users and maintainers.
- Main trick: **Keep it minimal!**
  - This means making choices that you would not make for a popular general-purpose language.

Conclusions

- **Designing a programming language for the desert** means coping with *persistent* scarcity of both users and maintainers.
- Main trick: **Keep it minimal!**
  - This means making choices that you would not make for a popular general-purpose language.
- **Realise that there are some things you just will not be able to afford.**
  - You might never have that advanced Language Server implementation.
  - So how can you design your language so someone can write a reliable go-to-definition tool in an afternoon?

Conclusions

- **Designing a programming language for the desert** means coping with *persistent* scarcity of both users and maintainers.
- Main trick: **Keep it minimal!**
  - This means making choices that you would not make for a popular general-purpose language.
- **Realise that there are some things you just will not be able to afford.**
  - You might never have that advanced Language Server implementation.
  - So how can you design your language so someone can write a reliable go-to-definition tool in an afternoon?

And why not go for a trip in the desert yourself?

`https://futhark-lang.org`