Meet **Beetroot**

CMake embedded language

that brings order to large deployments by introducing a **new paradigm** of work based on modularity and (real) **functions**

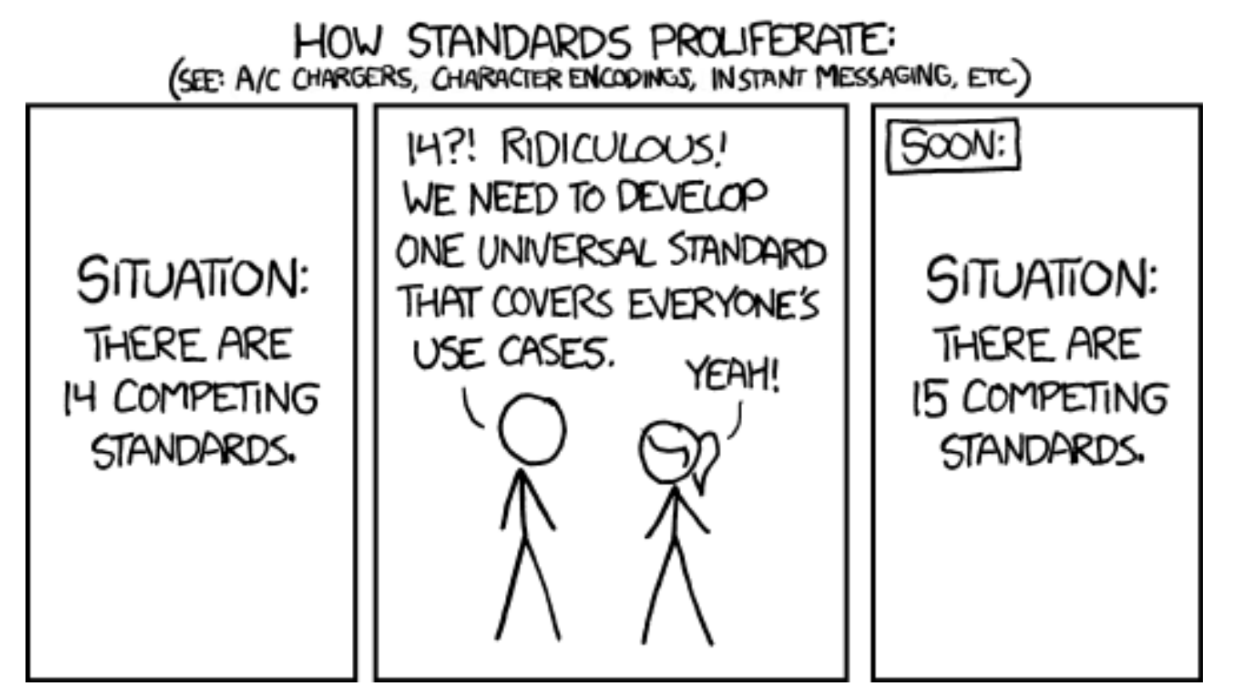
Summary

We propose The Beetroot, a project that structures user CMake code, enforcing deployments that are parametrizable and modular.

Beetroot is unique in the realm of the "total conversion mods" of CMake in that it does not replace CMake commands beyond `add_subdirectory()`, (which it discourages). **Knowledge of CMake is required to use the Beetroot.**

The Beetroot is a means to **parametrize build targets** together with their build dependencies and define a **clear API interface** for them. Thanks to that it offers **unparalleled flexibility in project design** and plenty of semantic checks on the user code.

It also supports easy and well-documented co-existance with pre-existing CMake code.


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

In order to avoid this situation, Beetroot does not replace/supersede CMake commands. Instead it allows for more efficient code structure. **Most of your code will stay intact - but in different places.**

Image from https://xkcd.com/927/

A little of CMake history

CMake is a language to describe a build process of a (usually C++) project. It was initially released in 2000 and among one of its advantages was an internal object structure that was focused on **build targets and their properties**. Since its start CMake was aiming to supersede the configure scripts and its own language drew heavily from the shell scripting including an unfortunate decision **not to include variable scope**. This design decision lead the user to use only global variables with no functions, which resulted in very poor modularization and hindered large deployments.

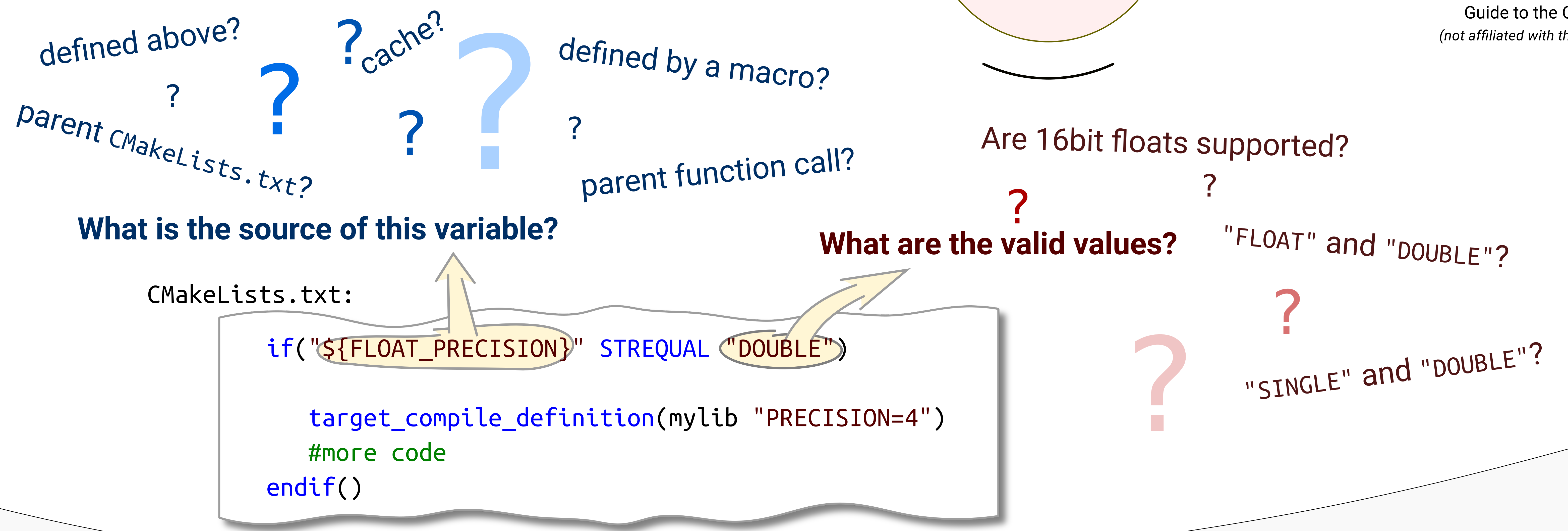
To mitigate this, KitWare added support for `add_subdirectory()` after CMake 2.0 and then `function()` with CMake 2.6. Support for named arguments came with CMake 2.8, as an afterthought to the already existing function command.

CMake continued to evolve, eventually introducing a so-called "**New CMake**" paradigm which focuses on attaching properties to targets (using set of specialized functions, like `target_compile_definitions()` and specify dependency with a single call to `target_link_libraries(DEPENDEE DEPENDENCY)`

Want to learn how to write efficient CMake code? Check out these two excellent CMake tutorials (***they are not affiliated with the Beetroot***):

CGold: The Hitchhiker's Guide to the CMake
(not affiliated with the Beetroot)

An Introduction to Modern CMake
(not affiliated with the Beetroot)



CMakeLists.txt:

```
if("${FLOAT_PRECISION}" STREQUAL "DOUBLE")

   target_compile_definition(mylib "PRECISION=4")
   #more code
endif()
```

Flexibility in API design

```
CMakeLists.txt
cmake_minimum_required(VERSION 3.13)
include(../beetroot/cmake/beetroot.cmake)

option(USE_GPU "Use GPU" OFF )
set(VERSION "0.9.8")

build_target(MY_LIB
   BUILD_TESTS
   COMPONENTS foo bar
   LINKPAR 13)

set(CPP_STD "cxx_std_14")
build_target(MY_LIB
   BUILD_TAG "cpp14"
   USE_GPU ON)

finalize()
```

Main features

- **Targets defined using Beetroot are formally parametrized.**
- Targets defined using Beetroot can be marked **as templates**, and the Beetroot will define a **distinct CMake target for each variation of its build parameters.**
- **Support for code that do not produce CMake targets.**
- **Distinction between build and link parameters**
- Beetroot tracks dependencies between its targets, **which can be parametrized.**
- **Beetroot does not replace existing CMake functionality.** Beetroot is about the macro scale, not micro: except for "`add_subdirectory()`" Beetroot does not try to replace or enhance existing CMake commands. In particular it does not interfere with the so-called "New CMake" paradigm.
- **Very flexible system of target parameters.** Passing in function call, passing by global value, default values... you get them all. Many types with type checking, including type-checked lists.

Even more features

- **Special handling for build parameters that describe incremental, extra or optional features.** Features can be flags, lists or version (e.g. API compatibility version). Target with the feature is assumed to be compatible with the base version.
- **Dependency can either define new (parametrized) target or already defined elsewhere target that is compatible with the given parameters or features.** The latter allows you to express the dependency as "*we require FOO with a parameter BAR and whatever other parameters that just happened to be set by whoever defined FOO elsewhere*".
- **Folder structure no longer needs to be influenced by the build process.** You can combine and build any parts of the project anywhere.

Cons

- **At least 2-3 times longer configuration time.** CMake is not really optimized for speed, and you may be surprised how **inefficient** variable handling in CMake is. Hopefully this will change in the future CMake releases.
- **Only tested for Linux.** Beetroot does not use any platform-specific CMake features, so in principle it should work on other platforms as well. We simply don't have a capacity to test it.
- **Extra dependency to maintain.**

Plans for the future

These are the features which implementation would easily fit into the Beetroot's data model and code structure

- **More data types. More checks on the arguments.**
- **Public repository of standard templates**, such as Boost, `configure_file`, **`generic_code_generator`,** `maybe_unittest`.
- **Support for code completion in common editors.**

Adam Ryczkowski
Institute of Meteorology and Water Management
National Research Institute

European Funds
Smart Growth

Foundation for Polish Science

European Union
European Regional Development Fund