QUAKE: BRIDGING THE BUILD SYSTEM GAP

Cassaundra Smith

whoami(1)

GOSIM 2024 EUROPE

~cassaundra

- Build system hacker
- Programming language enthusiast
- Former contributor to the Rust project
- Founder of Uredium Consulting
- Maintainer and developer of `quake`

Website https://cassaundra.dev
Email contact@cassaundra.dev
Fediverse @cassaundra@meow.lgbt
GitHub @cassaundra
sr.ht ~cassaundra

Key problem

Building UI applications is *hard*

UI app woes

- Cross-platform is already hard

UI app woes

- Cross-platform is already hard
  - Bundling makes it even harder (RPMs, APKs, Flatpak, macOS universal binaries, etc.)

UI app woes

- Cross-platform is already hard
  - Bundling makes it even harder (RPMs, APKs, Flatpak, macOS universal binaries, etc.)
- Language build systems don't care about anything but code in that language

Defining terms

What is a build system?

Defining terms

- **Compiler:** transforms sources into artifacts

Defining terms

- **Compiler:** transforms sources into artifacts
  - Examples: gcc, rustc, tar, pandoc (also: decompilers, transpilers)

Defining terms

- **Compiler:** transforms sources into artifacts
  - Examples: gcc, rustc, tar, pandoc (also: decompilers, transpilers)
- **Build system:** orchestrates units of compilation

Defining terms

- **Compiler:** transforms sources into artifacts
  - Examples: gcc, rustc, tar, pandoc (also: decompilers, transpilers)
- **Build system:** orchestrates units of compilation
  - Examples: make, ninja, Gradle, CMake, Buck2/Bazel/Pants

Build system taxonomy

Types of build systems:

Build system taxonomy

Types of build systems:

- **Task-based:** Executes tasks based on various conditions
  - Examples: `just`, `npm run`, most CI/CD systems

Build system taxonomy

Types of build systems:

- **Task-based:** Executes tasks based on various conditions
  - Examples: `just`, `npm run`, most CI/CD systems
- **Rules-based:** performs actions informed by source/artifact mappings
  - Examples: `make`, `ninja`, Meson (?), Buck2/Bazel/Pants

Build system taxonomy

Types of build systems:

- **Task-based:** Executes tasks based on various conditions
  - Examples: `just`, `npm run`, most CI/CD systems
- **Rules-based:** performs actions informed by source/artifact mappings
  - Examples: `make`, `ninja`, Meson (?), Buck2/Bazel/Pants
- **Domain-specific:** Designed for a given language/technology, benefits from domain-specific knowledge. Batteries included.
  - Examples: cargo, CMake, Maven, Meson

Build system taxonomy

Types of build systems:

- **Task-based:** Executes tasks based on various conditions
  - Examples: `just`, `npm run`, most CI/CD systems
- **Rules-based:** performs actions informed by source/artifact mappings
  - Examples: `make`, `ninja`, Meson (?), Buck2/Bazel/Pants
- **Domain-specific:** Designed for a given language/technology, benefits from domain-specific knowledge. Batteries included.
  - Examples: cargo, CMake, Maven, Meson
- **Custom:** Purpose-built systems for individual projects
  - `build.sh`, `cargo-xtask`, etc.

Build system taxonomy

Types of build systems:

- **Task-based:** Executes tasks based on various conditions
  - Examples: `just`, `npm run`, most CI/CD systems
- **Rules-based:** performs actions informed by source/artifact mappings
  - Examples: `make`, `ninja`, Meson (?), Buck2/Bazel/Pants
- **Domain-specific:** Designed for a given language/technology, benefits from domain-specific knowledge. Batteries included.
  - Examples: cargo, CMake, Maven, Meson
- **Custom:** Purpose-built systems for individual projects
  - `build.sh`, `cargo-xtask`, etc.

Example build systems

make(1)

- Initially developed in 1976, still widely used today
- [In]famously terse syntax, but powerful set of features
- Based on rules, which consist of targets, prerequisites, and a recipe
- Often generated by other tools

Example `Makefile`

```
CC=gcc
CFLAGS=-g

objects=foo.o bar.o

.PHONY: all
all: $(objects)

%.o: %.c
    $(CC) -c $(CFLAGS) -o $@

.PHONY: clean
clean:
    -rm -f *.o
```

Rule syntax

```
<targets>: <preqrequisites>
    <recipe>
    ...
```

Explanation

The target `all` has dependencies `foo.o` and `bar.o`.
These targets match the implicit `%.o: %.c` rule, which compiles C files into object files.

Advantages

- Rules system allows implicit optimization
- Recipes are shell scripts, a familiar construct
- On nearly every developer's machine
- Additional features allow for more powerful expression

Advantages

- Rules system allows implicit optimization
- Recipes are shell scripts, a familiar construct
- On nearly every developer's machine
- Additional features allow for more powerful expression

Disadvantages

- Difficult to read and write
- Brittle and hard to debug
- Shell script recipes ⇒ shell script problems

Bazel and Buck2

Bazel and Buck2 are in-house build systems that “own the world,” i.e. are aware of all files, dependencies, projects, etc.

Advantages

- Allows for highly optimized builds and customization
- Works well at scale

Bazel and Buck2

Bazel and Buck2 are in-house build systems that “own the world,” i.e. are aware of all files, dependencies, projects, etc.

Advantages

- Allows for highly optimized builds and customization
- Works well at scale

Disadvantages

- Requires a lot of boilerplate
- Poor interop with outside tooling (e.g. package managers)

Finding a better solution

Target usecases

Target usecases:

- Simple task runner
- Cross-platform application bundle generator
- Multi-stage build procedure where the native build system doesn't suffice

Build system requirements

What do we want out of a build system?

Build system requirements

What do we want out of a build system?

- **Expressive and flexible:** build scripts should be easy to *read*, *write*, and *debug*
  - Simple and complex build-time requirements should be both be easily expressed
  - Self-documenting, easily extensible, good error reporting

Build system requirements

What do we want out of a build system?

- **Expressive and flexible:** build scripts should be easy to *read*, *write*, and *debug*
  - Simple and complex build-time requirements should be both be easily expressed
  - Self-documenting, easily extensible, good error reporting
- **Transformation-aware:** understand source → artifact mappings
  - Inferred implicitly and/or from user input

Build system requirements

What do we want out of a build system?

- **Expressive and flexible:** build scripts should be easy to *read*, *write*, and *debug*
  - Simple and complex build-time requirements should be both be easily expressed
  - Self-documenting, easily extensible, good error reporting
- **Transformation-aware:** understand source → artifact mappings
  - Inferred implicitly and/or from user input
- **Language-agnostic:** support multilingual projects as a first-class feature

Build system requirements

What do we want out of a build system?

- **Expressive and flexible:** build scripts should be easy to *read*, *write*, and *debug*
  - Simple and complex build-time requirements should be both be easily expressed
  - Self-documenting, easily extensible, good error reporting
- **Transformation-aware:** understand source → artifact mappings
  - Inferred implicitly and/or from user input
- **Language-agnostic:** support multilingual projects as a first-class feature
- **Cross-platform:** both for the host, and for target platforms

Build system requirements

What do we want out of a build system?

- **Expressive and flexible:** build scripts should be easy to *read*, *write*, and *debug*
  - Simple and complex build-time requirements should be both be easily expressed
  - Self-documenting, easily extensible, good error reporting
- **Transformation-aware:** understand source → artifact mappings
  - Inferred implicitly and/or from user input
- **Language-agnostic:** support multilingual projects as a first-class feature
- **Cross-platform:** both for the host, and for target platforms
- **Hackable:**
  - Allow the system's simple rules to be faithfully abused
  - Produce machine-readable metadata for third-party tooling

Applying requirements

Specific improvements:

- Provide complete control over granularity
- Use modern languages better suited for the job
- Improve expressibility by reducing magic and boilerplate

Overall goal:

Ensure trivial cases are easy, and non-trivial cases scale at most linearly with their complexity

quake

What is quake?

`quake` is a cross-platform build system with build scripts written in a Nushell DSL.

Features:

- Declarative, self-documenting build script DSL
- Hybrid rule- and task-based build system
- Quality error reporting (thanks to `miette`)
- Powerful scripting and data manipulation (thanks to Nushell)

What is quake?

`quake` is a cross-platform build system with build scripts written in a Nushell DSL.

Features:

- Declarative, self-documenting build script DSL
- Hybrid rule- and task-based build system
- Quality error reporting (thanks to `miette`)
- Powerful scripting and data manipulation (thanks to Nushell)

Warning

Alpha code, not everything here works yet!

- Cross-platform, functionally influenced

- Cross-platform, functionally influenced
- Powerful data manipulation

- Cross-platform, functionally influenced
- Powerful data manipulation
  - Read, manipulate, and convert between JSON, TOML, etc. seamlessly

- Cross-platform, functionally influenced
- Powerful data manipulation
  - Read, manipulate, and convert between JSON, TOML, etc. seamlessly
  - Transform it through pipelines, FP/SQL style

Nushell example #1

Source: https://nushell.sh

```
/usr/bin 〉ls | where size > 10mb | sort-by modified
```

#	name	type	size	modified
0	x86_64-linux-gnu-lto-dump-10	file	23.3 MiB	a year ago
1	micro	file	13.7 MiB	8 months ago
2	buildah	file	19.8 MiB	7 months ago
3	qemu-system-i386	file	13.7 MiB	5 months ago
4	qemu-system-x86_64	file	13.7 MiB	5 months ago
5	node	file	76.6 MiB	a month ago

Nushell example #2

Source: https://nushell.sh

```
/home 〉 http get https://api.github.com/repos/nushell/nushell | get license
```

key	mit
name	MIT License
spdx_id	MIT
url	https://api.github.com/licenses/mit
node_id	MDc6TGljZW5zZTEz

def-task

Inside ***build.quake****:*

```
def-task say-hello [] run {
    echo "greetings!"
}
```

- Defines task named `say-hello`
- `run { ... }`: the **run block**
  - What the task performs when it is run

def-task (cont.)

```
def-task say-hello [] run {
    echo "greetings!"
}

def-task say-goodbye [] where {
    # declaration block
    depends-on say-hello
} run {
    # run block
    echo "goodbye!"
}
```

- Defines task `say-goodbye`, which depends on `say-hello`
- `where { ... }` **(declaration block)**
  - Contains declarative commands like **`dependency`**

Purely declarative tasks

We can also define tasks that are purely declarative:

```
def-task check-rustfmt [] do {
    cargo fmt --all-check
}

def-task check-clippy [] do {
    cargo clippy --workspace --all-features --all-targets -- -D warnings
}

# purely declarative--no `do` block!
def-task check [] where {
    depends-on check-rustfmt
    depends-on check-clippy
}
```

Task arguments

Tasks can take arguments!

```
def-task build [--release, package?: string, target?: string] {
    mut args = ["build"]

    if $release { $args ++= "--release" }
    if (not is-empty $package) { $args += ["--package", $package] }
    if (not is-empty $target) { $args += ["--target", $target] }

    provide $args # sets `$in` in the `run` block
} run {
    cargo ...$in
}
```

Conditionality

Everything is evaluated programmatically in Nushell

```
def-task build [] where {
    if $nu.os-info.name == "macos" {
        # note requires command
        requires "xcode toolchain is installed" check-xcode-toolchain
    }

    # ...
} run {
    # ...
}
```

Sources and artifacts

- Tasks have sources and artifacts
  - Represent a transformation
  - Declared with `sources` and `artifacts` respectively in the declaration block

Sources and artifacts

- Tasks have sources and artifacts
  - Represent a transformation
  - Declared with `sources` and `artifacts` respectively in the declaration block

```
 let crate_name = open Cargo.toml | get package.name

def-task build [] where {
    sources ["Cargo.{lock,toml}", "src/**/.rs"]
    artifacts [$"target/release/($crate_name)"]
} run {
    cargo build --release
}
```

More granular mappings

- More granularity is needed: introducing `transforms`
  - Adds a subtask with its own sources and artifacts

More granular mappings

- More granularity is needed: introducing `transforms`
  - Adds a subtask with its own sources and artifacts

```
let gcc_args = ["-g", "-O2"]

def-task build [] where {
    transforms ["foo.{c,h}"] into ["foo.o"] {
        # subtask run body
        gcc ...$gcc_args -c foo.c
    }

    transforms ["main.c", "foo.h"] into ["myprogram"] {
        gcc ...$gcc_args main.c foo.o
    }
}
```

Clean it up

- Granularity tends to add verbosity, so we should automate where we can.
- Declarative commands can be called in normal functions
  - ⇒ Write utility functions and toolchains!

Determining dependencies

Gathering metadata

```
$ clang -MM foo.c
foo.o: foo.c foo.h

$ clang -MT myprogram -MM main.c
myprogram: main.c foo.h
```

- `-M` commands are used for `make`-like rules already
- Works with many other languages (including Rust)

Determining dependencies

Gathering metadata

```
$ clang -MM foo.c
foo.o: foo.c foo.h

$ clang -MT myprogram -MM main.c
myprogram: main.c foo.h
```

- `-M` commands are used for `make`-like rules already
- Works with many other languages (including Rust)

Parsing with Nushell

```
~> clang -MM $target |
     parse "{target}: {deps}" |
     update deps { split row " " } |
     into record

+--------+---------------+
| target | foo.o         |
+--------+---------------+
|        | +---+-------+ |
| deps   | | 0 | foo.c | |
|        | +---+-------+ |
|        | | 1 | foo.h | |
|        | |   |       | |
|        | +---+-------+ |
+--------+---------------+
```

Clean it up

```
def build-target [target, binary?] {
    let result = if (is-empty $binary) {
        clang -MM $target
    } else {
        clang -MT $binary -MM
    }

    let rule = clang -MM $target |
        parse "{target}: {deps}" |
        update deps { split row " " } |
        into record

    transforms [$rule.target] into $rule.deps {
        # ...
    }
}
```

Conclusion

- Build systems need to catch up with language design

Conclusion

- Build systems need to catch up with language design
- Language designers need to do a better job exposing internals

Conclusion

- Build systems need to catch up with language design
- Language designers need to do a better job exposing internals
- Check out **quake** as it develops, or make something better!

Links

- **quake site:** https://quake.build
- **quake source:** https://github.com/quake-build/quake
- **Website:** https://cassaundra.dev
- **Email:** cass@cassaundra.dev
- **Fediverse:** @cassaundra@meow.lgbt
- **GitHub:** @cassaundra
- **sr.ht:** ~cassaundra

THANK YOU

GOSIM 2024 EUROPE