Jenga

The design of an expressive and scalable build system

What is a build system?

- Build rules provided by user
  - targets; actions; *dependencies*
- Build tool (`jenga`)
  - run actions necessary to bring targets up-to-date
- Shared framework (e.g. for a whole company)
  - per-project config
- Build process is demand driven

Why a new build system?

- Already so many to choose from:
  - make, omake, ocamlbuild, ninja, tup, redo, shake...
- Jane Street environment
  - 4k dirs; 34k files; 2.4m lines OCaml
  - Two workflows
- Focus: Correctness and Scalability

Design

Necessities

- Programmable
  - rule generation in a *real* programming language
  - `jengaroot.ml` dynamically compiled and loaded
- Incremental
  - *the* point of a build system
- Polling (inotify)
  - for individual development
- Parallel
  - run compilation actions in parallel

Incremental build

To build a target, locate its rule:

- Discover dependencies; bring them up to date
- Run the rule's action iff:
  - no record of running action before
  - dependencies have changed
  - action has changed
  - targets missing or different from expected
- Record successful run in persistent DB

Correctness

- Dependencies are tricky!
  - Accurate dependencies are required for consistent builds
  - Requires detailed knowledge of toolchain
- Dependencies can be dynamic
  - “scanner dependencies”
  - glob dependencies (`ls *.ml`)
- Rule generation
  - not a distinct phase
  - may also have dependencies

Jane Street

Rules `make` style

- Triple of targets, dependencies and action

```
val rule : path list -> dep list -> action -> rule
```

- Not expressive enough!
  - Can't represent *dynamic dependencies*
  - Action is fixed

Jane Street

Encoding dependencies

Introduce a notion of *a value and its dependencies*.

α `dep`

Constant value:

```
val need     : path -> unit dep
```

Varying value:

```
val glob     : dir:path -> string -> path list dep
val contents : path -> string dep
```

Jane Street

Composing dependencies

Dynamic dependencies expressed with `bind ( *>>= )`

```
val return   : 'a -> 'a dep
val ( *>>= )  : 'a dep -> ('a -> 'b dep) -> 'b dep
val ( *>>| )  : 'a dep -> ('a -> 'b    ) -> 'b dep
```

Concurrency expressed using `all`

```
val all      : 'a dep list -> 'a list dep
val all_unit : unit dep list -> unit dep
```

Jane Street

Rules `jenga` style

- Action carried by the dependency

```
val rule : path list -> action dep -> rule
```

- Rule generation

```
val generate : (dir:path -> rule list dep) -> scheme
```

- Recover simple rules

```
let simple_rule targets deps action =
  rule targets (
    all_unit deps *>>= fun () ->
    return act)
```

Jane Street

Example 1: OCaml compilation

```
val compile_ml: dir:path -> name:string -> rule

let compile_ml ~dir ~name =
  let p x = relative ~dir (name ^ x) in
  rule [p".cmi"; p".cmx"; p".o"] (
    let static = [p".ml"] in
    deps_from_file ~dir (p".ml.d") *>>= fun dynamic ->
    needs (static @ dynamic) >>| fun () ->
    bash ~dir (sprintf "ocamlopt -c %s.ml" name)
  )
```

Jane Street

Example 2: OCaml rule generation

```
val generate : (dir:path -> rule list dep) -> scheme
```

Rules for a directory of ocaml

```
generate (fun ~dir ->
  glob ~dir "*.ml" *>>= fun mls ->
  glob ~dir "*.mli" *>>| fun mlis ->
  let exists_mli x = List.mem mlis (relative ~dir (x ^ ".mli")) in
  List.map mls ~f:(fun ml ->
    let name = chop_suffix (basename ml) ".ml" in
    if (exists_mli name)
    then compile_ml_mli ~dir ~name
    else compile_ml ~dir ~name)
)
```

Summary of Jenga

- Key features
  - Rule development in OCaml
  - Expressive API for dynamic dependencies
  - *Incremental*, *polling*, *parallel* builds
- Developed and used at Jane Street
- Open source

```
opam install jenga
```

Jane Street