Using Memo in Dune

What is Memo?

What is Memo?

- an incremental computation library
- an "in-memory build-system"

Why Memo?

Why Memo?

To speed-up incremental builds in polling mode.

Memo delivers!

Jane Street code base:

- before: tens of seconds
- after: a few seconds

What is Memo about?

What is Memo about?

Memoising pure functions.

- during a single build
- during rebuilds

Implicit dependencies

- between function calls
- recorded rather than declared

But

- src/dune_rules is not pure yet

The API

The Memo.Build monad

- can only call memoised function inside
- the main monad
- replaces fibers

Memo.create

```
let build_file : Path.t -> Digest.t Memo.Build.t =
  let memo =
    Memo.create "build_file"
      ~input:(module Path)
      (fun path ->
        <locate rule for path>
        <execute this rule>)
   in
   fun path -> Memo.exec memo path
```

cutoffs

```
let build_file : Path.t -> Digest.t Memo.Build.t =
  let memo =
    Memo.create "build_file"
      ~input:(module Path)
      ~cutoff:Digest.equal
      (fun path ->
        <locate rule for path>
        <execute this rule>)
   in
   fun path -> Memo.exec memo path
```

lazy

```
let x = Memo.lazy_ (fun () -> ...) in
...
Memo.Lazy.force x
```

Compared to plain lazy:

- caches the `'a` rather than the `'a Build.t`

Working with Memo.Build

case study: fold

```
let collect_libs () : Lib_set.t =
  Source_tree.fold ~init:Lib_set.empty ~f:(fun dir acc ->
    Lib_set.union acc <libs-in-dir>)
```

parallel-friendly

```
let collect_libs () : <lib-set> =
  Source_tree.map_reduce (module Monoid.Union(Lib_set))
    ~f:(fun dir -> <libs-in-dir>)
```

Memo call stack and error messages

Dependency path

```
$ dune build --debug-dependency-path
File "dune", line 1, characters 0-47:
1 | (rule
2 |  (action (with-stdout-to x (run false))))
       false x (exit 1)
(cd _build/default && /bin/false) > _build/default/x
-> required by _build/default/x
-> required by _build/default/y
-> required by alias default in dune:6
```

human_readable_description

```
val create :
     string
  -> input:(module Input with type t = 'i)
  -> ?cutoff:('o -> 'o -> bool)
  -> ?human_readable_description:
       ('i -> User_message.Style.t Pp.t)
  -> ('i -> 'o Build.t)
  -> ('i, 'o) t
```

Works for cycles as well

- not just between files
- alternative to enabled_if restrictions

Memo.dump_stack

- equivalent of `Printexc.get_call_stack ... |> <print>` for the memo stack
- contains inputs

What's next?

Future

- faster
- better debugging and profiling
- persistence?

Questions?