Buck2 for OCaml Users and Developers

Shayne Fletcher Neil Mitchell

Meta Platforms Inc.

shaynefletcher@meta.com

OCaml Users and Developers Workshop
August 10, 2023

Roadmap

Buck2

A multi-language large-scale build system open-sourced by Meta
`https://buck2.build`.

Written in Rust over the last 4 years by a team of people.

Multi-language

- Buck2 has no baked in knowledge of *any* programming language.
- Configured through Starlark/Python files which say how to build `ocaml_library` etc.
- Rules produce *providers* that say how they provide stuff.
  - E.g. all *native* languages produce `MergedLinkInfo`.
  - OCaml produces it, Rust can work with it, the system linker can link them.
  - Therefore, C++ can depend on Rust which depends on C++.
- More generally, languages don't need to know about OCaml to link with it.

Large scale builds

- Buck2 is designed for large scale (millions of files).
- File watching with `watchman` - too many files to check modification time.
- Bazel compatible remote execution.
  - If anyone else has already run a command, just copy.
  - Run commands remotely on a server - thousands at a time.
- Deferred materialisation - if an intermediate product is available remotely, don't download it.

Theoretical power

- Provides monadic/dynamic dependencies as per Build Systems à la Carte [1].
- An OCaml library must have its files compiled in *dependency order*.
- Buck1: Run `ocamldep` once and hope it doesn't change much.
- Bazel: specify the internal file dependencies.
- Buck2: runs `ocamldep` automatically and follow the dependencies.
  - Define the OCaml library dependency node and declare it outputs a `.cmxa`.
  - Run the `ocamldep` tool, producing a text file (`Makefile`).
  - Read the output, parsing it (in Starlark) to produce a graph.
  - Fill OCaml compilation commands into that graph.
  - Point at where the output file ends up.

Roadmap

Buck2 OCaml Examples

The referenced examples are from the facebook/buck2 GitHub repository[a].

[a] See the examples/with-prelude/ocaml directory.

Hello world

Example (Library)

```
# build with: buck2 build //ocaml/hello-world:hello-world-lib
ocaml_library(
  name = "hello-world-lib",
  srcs = [ "hello_world_lib.ml" ],
)
```

Example (Binary)

```
# build & run with: buck2 run //ocaml/hello-world:hello-world --
ocaml_binary(
  name = "hello-world",
  srcs = [ "hello_world.ml" ],
  deps = [ ":hello-world-lib" ],
)
```

Bytecode vs. native

Use the `bytecode` sub-target to produce OCaml programs built via `ocamlc`:

- Run native executable
  - `buck2 run ':hello-world'`
- Run bytecode (standalone) executable
  - `buck2 run ':hello-world[bytecode]'`

Use `--show-output` to locate materialized artifacts:

- `buck2 target ':hello-world' --show-output`
  - `buck-out/.../hello_world/__hello-world__/hello-world.opt`
- `buck2 target ':hello-world[bytecode]' --show-output`
  - `buck-out/.../hello_world/__hello-world__/hello-world`

Native rules

The full set of Buck2 prelude OCaml rules:

- `prebuilt_ocaml_library` ('.cma', '.cmxa')
- `ocaml_library` ('.cma', '.cmxa')
- `ocaml_binary` ('.opt' or no extension)
- `ocaml_object` ('.o')
- `ocaml_shared` ('.cmxs')[1]

[1] Native code plugin suitable for use with the Dynlink module

Integrating OPAM

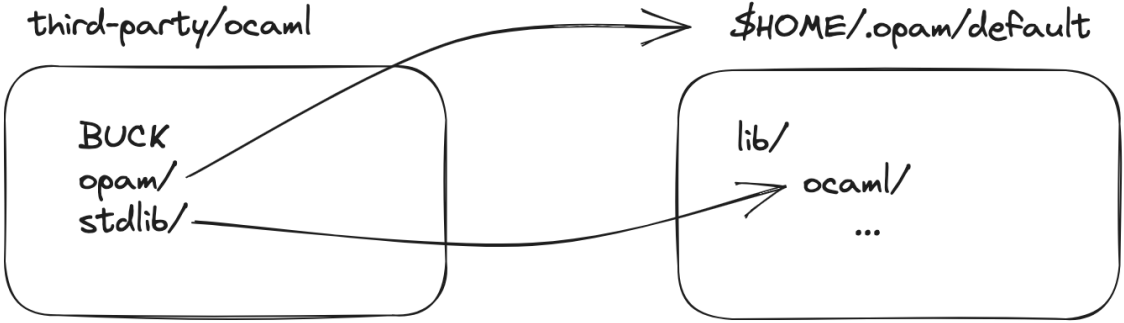

Figure: Symlinks into .opam

2

[2]Scripts to help OCaml projects using Buck2 facebook/ocaml-scripts.

Prebuilt libraries

Example (Defining a prebuilt library)

```
prebuilt_ocaml_library(
    name = "ppxlib",
    include_dir = "opam/lib/ppxlib",
    native_lib = "opam/lib/ppxlib/ppxlib.cmxa",
    ...
)
```

Example (Using a prebuilt library)

```
ocaml_library(
    name = "ppx-record-selectors",
    deps = [ "//third-party/ocaml:ppxlib", ... ],
    ...
)
```

Parsers, lexers and interfacing with C

Example (Using ocamllex, menhir)

```
# build & run with: buck2 run //ocaml/calc:calc
ocaml_binary(
    name = "calc",
    srcs = [ "calc.ml","lexer.mll", "parser.mly", ],
)
```

Example (Interfacing with C)

```
ocaml_binary(
    name = "...",
    srcs = [ "fib.ml", "fib.c", ],
)
```

Defining a Ppx

Example (Define 'record selectors')

```
ocaml_library(
    name = "ppx-record-selectors",
    srcs = ["record_selectors.ml"],
    deps = [ "//third-party/ocaml:ppxlib" ],
)
ocaml_binary(
    name = "ppx",
    srcs = ["ppx_driver.ml"],
    compiler_flags = [ "-linkall" ],
    deps = [ ":ppx-record-selectors", ],
)
```

```
92 let impl_generator = Deriving.Generator
93 let intf_generator = Deriving.Generator
94 let _ = Deriving.add "record_selectors"
   type_decl:intf_generator
95
```

Figure: 'record_selectors.ml'

```
10 let () = Ppxlib.Driver.standalone ()
```

Figure: 'ppx_driver.ml'

Using a Ppx

```
10 type t = {
11   foo: int;
12   bar: string;
13 }
14 [@@deriving record_selectors]
15
16 let r: t = { foo = 4; bar = "quux" }
17 let (): unit = Printf.printf "%d %s\n" (foo r) (bar r)
```

Figure: 'ppx_record_selectors_test.ml'

Inspecting preprocessed source

Use the 'expand' sub-target to make the elaborated program text available for inspection (e.g. `buck2 build //ocaml/ppx:'ppx-record-selectors-test[expand]'`).

Example (Use 'record selectors')

```
ocaml_binary(
    name = "ppx-record-selectors-test",
    srcs = [ "ppx_record_selectors_test.ml" ],
    compiler_flags = [ "-ppx", "$(exe_target :ppx) --as-ppx" ],
)
```

Embedding

Example (OCaml)

```
ocaml_object(
  name = "fib-ml",
  srcs = ["fib.ml"]
)
```

Example (C++)

```
cxx_binary(
  name = "fib-cpp",
  srcs = ["fib.cpp"],
  deps = [ ":fib-ml", ... ],
)
```

User defined primitive written in OCaml...

```
10 let rec fib n =
11   if n < 2 then
12     1
13   else
14     fib (n - 1) + fib (n - 2)
```

Figure: 'fib.ml'

... linked with and called from C++.

```
18 int fib(int n) {
19   static const value* fib_closure = NULL;
20   if (fib_closure == NULL)
21     fib_closure = caml_named_value("fib");
22   return Int_val(caml_callback(*fib_closure, Val_int(n)));
23 }
```

Figure: 'fib.cpp'

Extending

User defined primitive, written in Rust...

```
10 #[no_mangle]
11 pub unsafe extern "C" fn caml_print_hello(_: usize) -> usize {
12     println!("Hello Rust!\n");
13     return 0;
14 }
```

Figure: 'hello_stubs.rs'

... linked with and called from OCaml.

```
10 external print_hello: unit -> unit = "caml_print_hello"
11
12 let () = print_hello ()
```

Figure: 'hello.ml'

Example (Rust)

```
rust_library(
    name = "hello-stubs-rs",
    srcs = [ "hello_stubs.rs" ],
)
```

Example (OCaml)

```
ocaml_binary(
    name = "hello-rs",
    srcs = [ "hello.ml" ],
    deps = [":hello-stubs-rs"],
)
```

mylib

'mylib.mli': alias map

```
10 module A = Mylib__A
11 module B = Mylib__B
```

'mylib__A.ml' implements `A`

```
10 let print_hello () = B.print_hello ()
```

'mylib__B.ml' implements `B`

```
10 let print_hello () = Printf.printf "Hello world!\n"
```

Exercising mylib functions requires qualified syntax ('test_Mylib.ml'):

```
10 let _: unit = Mylib.A.print_hello ()
```

mylib targets

```
export_file(name = "mylib.mli", src = "mylib.mli")

ocaml_library(
    name = "mylib__",
    srcs = ["mylib.ml", ":mylib.mli"],
    compiler_flags = ["-no-alias-deps", "-w","-49"],
)

ocaml_library(
    name = "mylib",
    srcs = ["mylib__A.ml", "mylib__B.ml"],
    compiler_flags = ["-no-alias-deps", "-w","-49", "-open","Mylib"],
    ocamldep_flags = ["-open","Mylib", "-map","$(location :mylib.mli)"],
    deps = [":mylib__"],
)
```

Roadmap

Pyre

Pyre is a typechecker for Python with $\approx$ 300 files. With Buck2, a build can be obtained from a full remote cache in $\approx$ 12*s*. Tests on a 72 core VM:

Tool	Time	RSS
Dune	4m25s	377KB
Buck2	3m09s	180KB

Table: Dev, single thread

Tool	Time	RSS
Dune	0m51s	377KB
Buck2	0m33s	180KB

Table: Dev, default thread settings

Tool	Time	RSS
Dune	7m54s	480KB
Buck2	7m11s	180KB

Table: Release, single thread

Tool	Time	RSS
Dune	3m56s	377KB
Buck2	4m23s	178KB

Table: Release, default thread settings

Flow

Flow is a multi-purpose binary for JavaScript language services with $\approx$ 1000 files. With Buck2, a build can be obtained from a full remote cache in $\approx$ 12*s*. Tests on a 72 core VM:

Tool	Time
Dune	4m38s
Buck2	6m33s

Table: Dev, single thread

Tool	Time
Dune	0m41s
Buck2	0m59s

Table: Dev, default thread settings

Tool	Time
Dune	4m56s
Buck2	9m33s

Table: Release, single thread

Tool	Time
Dune	1m35s
Buck2	2m42s

Table: Release, default thread settings

Roadmap

Buck2

Meta Open Source

Roadmap

References


Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones.
Build systems à la carte.
In *Proceedings of the ACM on Programming Languages, Volume 2 Issue ICFP*, 2018.