Jbuilder: a modern approach to OCaml development

Jeremie Dimino and Mark Shinwell, Jane Street Europe

Jbuilder is the first tool to provide a seamless workflow for the development of OCaml projects. Straightforward and easy-to-learn, it abstracts away from typical complexities of build and package description systems, at the same time helping to maximise portability. We argue that Jbuilder has the potential to unify the OCaml community and to increase the productivity of developers.

Motivation

The development of OCaml projects for public release typically involves three phases:

- **Compilation:** producing libraries and executables from source files.
- **Package layout:** describing where the software components will appear when installed.
- **Packaging and distribution:** creating release tarballs and publishing them.

It is now commonplace for projects to be published to the global repository used by the `opam` package manager. This makes it easy for end users to find, download and install software onto their machines. However getting to the stage of published packages given source files is still a relatively arduous process.

The compilation and layout steps currently require the use of a patchwork of different tools. These range from build systems such as `make` and `ocamlbuild` through to helpers for the build process such as `ocamlfind` (used to locate libraries already installed on the system) and tools that assist with package layout such as `oasis`. These various tools may work well individually and can be made to work together. However there are many ways of assembling them into a moderately coherent whole especially given the flexibility afforded to build systems by the OCaml compiler. This, coupled with the fragmented nature of the development of these tools, has meant that there exists no canonical documented workflow that works well for the majority of projects.

There is some evidence that this situation causes new users of the language to be disenchanted from the beginning. Experienced users also suffer, having to spend their time maintaining bespoke build and package layout systems, instead of writing code and documentation. Such systems are rarely future proof and they are often not as portable as the code itself. Problems of non-portability are exacerbated by the fact that most OCaml developers use Unix-like platforms: support for Microsoft Windows often ends up below par, for example. The lack of uniformity is also not conducive to large-scale analysis or testing for research purposes.

Description

Jbuilder provides a simple declarative language which developers may use to describe their projects in terms of libraries and executables together with auxiliary items such as generated documentation and automated test cases. Unlike the inputs to existing tools this language is sufficient to implement the whole compilation and package layout workflow. The packaging and distribution steps are left to a third-party tool called `topkg`; this currently requires minimal additional configuration and is planned to be more tightly integrated with Jbuilder in due course. The aim is that no configuration beyond that provided to Jbuilder will be required in order to publish packages to the `opam` repository.

Jbuilder abstracts away from the low-level details of OCaml compilation; yet it makes full use of recent developments such as namespace isolation across libraries (using module aliases, not packed modules). This relieves developers of a significant burden. Jbuilder enforces certain conventions (for example that the `opam` package name matches the `ocamlfind` package name) with a view to increasing consistency across packages.

Jbuilder also provides sufficient functionality for the majority of projects to be built without any need for an external shell. This is achieved in two ways. Firstly, high-level constructs in the description language provide core functionality (in itself sufficient for a substantial number of projects) such as the compilation of OCaml source files into a library. Secondly, operations are provided corresponding to those that typically might be used via a shell; these may be used to form custom build rules when the higher-level constructs are not sufficient. (An example might be running a program that auto-generates source files.) These operations will work correctly on all supported platforms including Microsoft Windows (even without Cygwin).

Jbuilder by default performs correct dependency analysis and runs build steps in parallel. This, combined with the lack of external shell calls and an efficient internal scheduling mechanism, produces fast build and rebuild times.

It is expected that support for cross compilation, subject to the necessary additions to the OCaml compiler, will be straightforward to implement in Jbuilder. Even at present, Jbuilder supports a notion of build context, an abstraction that enables a package to be simultaneously built against multiple versions of the compiler.

The description language used by Jbuilder has been designed to minimise complexity. This helps not only to make it understandable by users but also by tools. Large-scale analysis of build and packaging instructions by machine across multiple projects can now be conducted. An element of future proofing is also inherent: automatic upgrading to either future versions of Jbuilder (were there ever a need to break backwards compatibility in the description language) or some hypothetical future tool should be straightforward. At the very least it is feasible.

Jbuilder itself is written entirely in OCaml. It depends only on the standard library and the `Unix` library found in the OCaml distribution. The build process of Jbuilder uses a script written in OCaml; as such it does not depend on any tool such as `make`.

Evaluation and future work

Since the beta release of Jbuilder in March 2017 several hundred packages, from a variety of different authors and organisations, have been converted to it. Feedback from these conversions has been overwhelmingly positive. For example, the MirageOS project has migrated various of its libraries, with build times for some of them being reduced by nearly tenfold.

Jbuilder development has so far been focussed on establishing a solid foundation for the most common tasks. The functionality thus produced has been shown to be sufficient for a multitude of projects. However feedback from the community indicates that some projects have bespoke requirements that cannot easily, at present, be satisfied inside the Jbuilder language. One main avenue of future work will be to decide whether and how to make Jbuilder more extensible to support such use cases. Extensibility could perhaps take the form of some kind of plugin system using an OCaml API. However a balance needs to be struck between the expressiveness provided by such extensibility and the complexities, for example surrounding stable programming interfaces, that might arise. Unconstrained extensibility, despite being attractive for users, might not be the best thing for either Jbuilder or the community in the long term.

Not three months after the beta release, external contributions to Jbuilder itself are already starting to arise, with much associated discussion. It is fully expected that these will continue and increase.

Conclusion

Jbuilder was in fact initially designed to only solve a narrow problem within one particular company: the problem of releasing a large number of public packages from a private code base at Jane Street. However, it soon became clear that the technology developed for this purpose could be more widely applicable. The initial widespread adoption of the tool has been pleasing; it would seem to serve as a good example of industrial collaboration with a community of language users.

Our thesis is that Jbuilder has the potential to become the canonical means for the development of OCaml packages. It is hoped that substantial unifying benefit to the community would follow.