Programming Unikernels in the Large via Functor Driven Development (Experience Report)

ANONYMOUS AUTHOR(S)

Compiling applications as unikernels allows them to be tailored to diverse execution environments. Dependency on a monolithic operating system is replaced with linkage against libraries that provide specific services. Doing so in practice has revealed a major barrier: managing the configuration matrix across heterogenous execution targets. A realistic unikernel application depends on hundreds of libraries, each of which may place different demands on the different target execution platforms (e.g., cryptographic acceleration).

We propose a modular approach to structuring large scale codebases that cleanly separates configuration, application and operating system logic. Our implementation is built on the MirageOS unikernel framework, using the OCaml language's powerful abstraction and metaprogramming facilities. Leveraging modules allows us to build many components independently, with only loose coupling through a set of standardised signatures. Components can be parameterized by other components and composed. Our approach accounts for state, dependency ordering, and error management, and our usage over the years has demonstrated significant efficiency benefits by leveraging compiler features such as global link-time optimisation during the configuration process. We describe our application architecture and experiences via some practical applications of our approach, and discuss how library development in MirageOS can facilitate adoption in other unikernel frameworks and programming languages.

Additional Key Words and Phrases: MirageOS, unikernels, functional, modules, OCaml

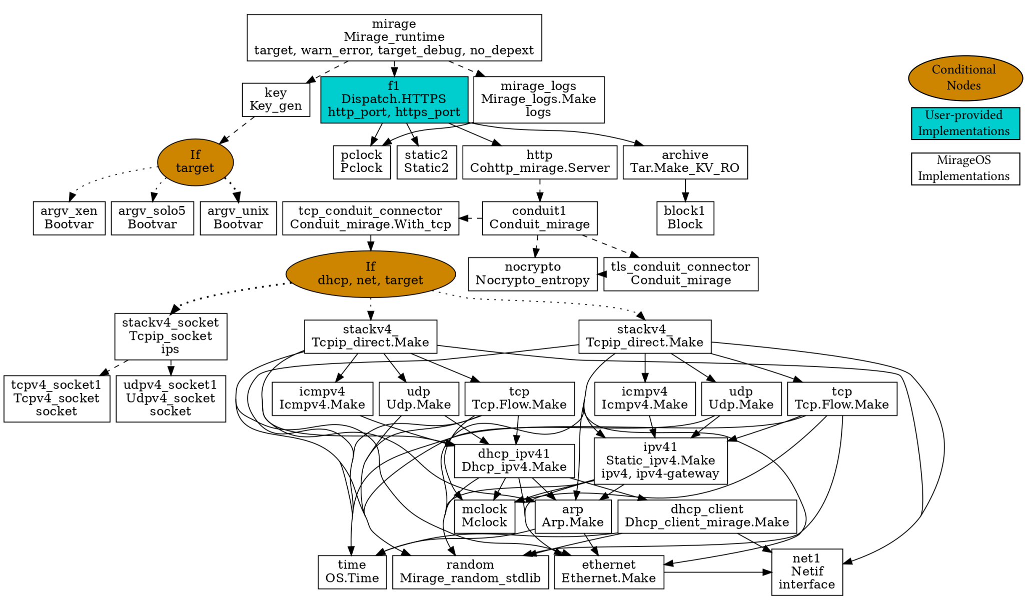

Fig. 1. Configuration graph for a MirageOS web server

2475-1421/2019/1-ART1 $15.00
https://doi.org/

1 INTRODUCTION

A major source of complexity in modern application development is the need to run on an increasingly diverse range of platforms: conventional operating systems (OSs) such as Linux or Windows, mobile systems such as Android or iOS, embedded platforms such as ARM or RISC-V microcontrollers, or browser-based virtual machines via compilation to JavaScript or WASM. Designing efficient programming interfaces for such heterogenous environments is challenging as all have different internal models and mechanisms for memory management, isolation, I/O and scheduling.

Attempts to adapt existing models (e.g. POSIX) to these environments has led to a lowest common denominator set of "mini-libc" system libraries. These are deeply unsatisfying: one would rather generate binaries that are specialised for a particular platform, able to make full use of its specific capabilities. Ideally, we would have a modular set of interfaces allowing applications to depend on the specific functionality they need to operate on a specific physical or virtual platform.

One key step towards this goal is to use library operating systems (libOSs) to break down monolithic kernel components into conventional libraries that can be linked alongside application logic [Engler et al. 1995; Leslie et al. 1996]. When these kernel and application libraries are linked to a bootloader, the result is a single-purpose *unikernel*, specialised at build time to execute that specific application on the specific platform [Madhavapeddy et al. 2013]. The specialisation has been shown to result in significant performance and code-size improvements in the resulting artefacts [Madhavapeddy et al. 2015; Manco et al. 2017].

The last few years have seen many new unikernel frameworks written in high-level languages. The number of kernel libraries has grown concomitantly, resulting in a practical challenge: *how can developers avoid the need to manually select the set of kernel libraries required by a target platform*? The common approach of depending on a monolithic OS interface layer (e.g., in OCaml, the `Unix` module), does not scale to the modern heterogeneous world.

This paper describes our experiences in addressing this problem of writing high level code that can run in heterogenous execution environments by using OCaml's powerful abstraction facilities within the MirageOS unikernel framework. MirageOS has been developed since 2006 and has seen widespread deployment in industrial projects such as Xen [Gazagnaire and Hanquez 2009; Scott et al. 2010] and Docker. Over the last decade, MirageOS has grown to support a highly diverse set of target platforms including hypervisors such as Xen [Barham et al. 2003], KVM [Kivity et al. 2007] and Muen [Buerki and Rueegsegger 2013], plus conventional Unix and Windows binaries, and even experimental compilation to JavaScript and bare-metal booting on RISC-V and ARM boards.

The key challenge in maintaining these compilation targets has been to prevent OCaml programmers, otherwise fastidious about their use of abstraction, from using the monolithic OS interfaces such as `Unix` that tie an application to a single execution environment. Instead, MirageOS takes advantage of the powerful ML module system to allow programmers to abstract over use of individual OS facilities (e.g., timekeeping, networking, storage, entropy). Rather than calling into `libc`, application code is abstracted over the OS functionality needed using OCaml's parameterised modules. The MirageOS compiler then supplies library implementations of the required functionality suitable for the target platform. These implementations range from trivial passthroughs that invoke system calls on Unix, to complete reimplementations of key kernel subsystems such as TCP/IP for targets without a conventional OS such as bare-metal embedded devices or Xen hypervisors.

This way, developers write application code that can be efficiently compiled to any of these environments simply by making their dependencies on system facilities explicit using parameterisation. The resulting codebases are also highly structured (see Figure 1 for the MirageOS webserver) and easily compiled to future deployment targets. We dub this approach *Functor Driven Development*, and make the following experience contributions in this paper:

- we describe our portable application structuring that encourages developers to explicitly specify OS dependencies by using OCaml module constructs: structures, signatures, and functors (i.e. functions over modules) (Section 2);
- we show how we make use of meta-programming techniques to generate the complex glue code that connects configuration, build and deployment of the application, using an eDSL to express dependencies between the application requirements and concrete implementations for a particular target platform (Section 3); and
- discuss our experiences with using the OCaml module system at scale for operating system assembly (Section 4).

2 STRUCTURING APPLICATIONS WITH FUNCTORS

OCaml modules [Leroy 1994, 1995; MacQueen 1984] allow programs to be built from smaller components. In most languages, modules are compilation units: simple collections of type and value declarations in a file. OCaml extends such collections, called *structures*, with *signatures* (module types), *functors* (functions from modules to modules) and functor application, to form a small typed functional language. Developers use this language to group, compose and selectively expose program components (types, values, functions, and modules). Modules are structurally typed: a module need not announce which signatures it satisfies, and a single module can satisfy many different signatures, which may expose or conceal module components, and present types as concrete or abstract. Modules may be combined using functors, which construct new modules from existing modules passed as arguments.

Figure 2 uses this technique to design a simple static file server with two module parameters: `S` of type `Store`, which describes how to access local files, and `N` of type `Network`, which describes how networking is managed. `Store` and `Network` each expose a type `t` (representing the storage and

```
module type Store = sig
  type t
  val read: t -> string -> string
end
module type Network = sig
  type t
  val listen:t -> (string -> string) -> unit
end
```

```
module Make (S: Store) (N: Network) = struct
  let start storage network =
    N.listen network (S.read storage)
end
```

Fig. 2. A modular file server.

```
module Direct : sig
 (* read host filesystem *)
 include Store
 val create : string -> t
end
```

(a) `Direct` implements `Store`.

```
module Crunch : sig
 (* read compiled-in strings *)
 include Store
 val create : Crunch.t -> t
end
```

(b) `Crunch` implements `Store`.

```
module NetStore (N: Network): sig
 (* read from network service *)
 include Store
 val create : N.t -> t
end
```

(c) `NetStore` implements `Store`

Fig. 3. Examples of store implementations

```
module TCPIP : sig
  include Network
  val create : int -> t
end
```

(a) `TCPIP` implements `Network`.

```
module HTTP (N : Network) : sig
  include Network
  val create : N.t -> t
end
```

(b) `HTTP` implements `Network`.

Fig. 4. Examples of network implementations

network handles respectively) and a function: `listen` makes a callback that listens on the network handle, and `read` accesses the current store to read a file. The core application logic is defined by the functor `Make` whose body contains a single function that calls the (abstract) functions from its module parameters `N` and `S`.

Figure 3 and Figure 4 show several storage and network implementations. `Direct` (Figure 3a), `Crunch` (Figure 3b) and `NetStore` (Figure 3c) implement various kinds of `Store`. As well as satisfying the signature `Store`, each implementation also provides a `create` function with specialised arguments to take care of device-specific initialization. `Direct.read` gives access to the underlying filesystem, the handle being the root of the filesystem in question. `Crunch` provides an in-memory representation of a file-system. It operates by turning a filesystem tree into an OCaml module which is then compiled and embedded in the application at configuration time. Finally, `NetStore` presents an online service as an initially-empty `Store`; it processes requests to add files. `NetStore` requires network access and is thus a functor parameterised by a module of type `Network`.

`TCPIP` (Figure 4a) and `HTTP` (Figure 4b) implement `Network`. The function `TCPIP.listen` uses the POSIX `listen` and `accept` syscalls to handle incoming TCP/IP connections on the given port. It then reads a request line and returns the result of passing it to a callback. The function `HTTP.listen` handles connections, reading a full HTTP request when a client connects, extracting the HTTP path and passing it to the callback. The resulting file content is wrapped into an HTTP response by adding the correct headers, before being returned to the client connection. Note that this implementation depends on another network stack to simply read request and response contents without interpretation. We can use this to implement HTTP over TCP/IP or over TLS to get HTTPS.

Each of these modules can be used to satisfy the application's functor allowing our simple static fileserver to target a very wide range of deployment platforms. In each implementation, the type `t` represents wildly different states but, as `t` is abstract, OCaml ensures that details of the type's implementation are never used in the body of the `Make` functor in Figure 2.

2.1 Standardized Signatures

Our example application consists of two major pieces of external functionality: file system access and networking. MirageOS separates these two domains from the usual monolithic `Unix` module by defining independent module signatures, which are then implemented by several modules. This modular approach has two advantages: it avoids a dependency on a monolithic OS kernel, and it disaggregates functionality into smaller module signatures that can be separately implemented by experts in each domain. File system experts can contribute implementations of the `Store` signature, and network developers can write `Network` implementations. The signature approach also makes dependencies between different domains explicit; for example, the `NetStore` implementation interacts with both `Network` and `Store`.

This strong isolation of concerns has proven essential in growing the MirageOS ecosystem. An operating system contains many pieces pertaining to very different domains. MirageOS contains libraries ranging from bare-metal drivers to TLS implementations, including high-level HTTP servers. Contributors' knowledge in a given domain can be applied to build additional implementations that will fit into the overall ecosystem, without getting overwhelmed by the enormity of the full clean-slate operating system stack.

Having implementations bundled as modules with a common interface is also beneficial for testing purposes. Complex components can be tested in isolation and often without requiring a physical environment. Tests can be expressed as functors over the signatures to test, allowing us to stress the implementation in a virtual environment convenient for local use (e.g. a fake networking bridge). We also use this approach to test the applications themselves, which are also parameterised by their module dependencies. We have combined this parameterised testing approach with property testing [Claessen and Hughes 2000] and fuzzing [Dolan and Preston 2017; Zalewski 2014] in various implementations.

2.2 State and Initialization

All the functors and modules in the previous sections are *pure*: they do not produce side effects when applied to other modules. Applying a functor creates a new module built from its parameters, but does not perform initialization or modify state. This is convenient for two reasons. First, modules might share an interface for most of their operations except for the initialization code. For example, in our store implementations (Figure 3) the type of `create` varies with each implementation, but besides `create` the implementations all simply implement the `Store` signature. By separating initialisation functions from the rest of the operations, we ensure that the core application, such as the `Make` module in Figure 2, can be used with a large variety of implementations. Second, purity maximises implementation sharing without mixing up state. For example, in the `NetStore` module we might share the same `Network` implementation for both the online repository and to serve files. However, although the implementations are shared, the network handle itself is not, ensuring we don't accidentally couple the two otherwise-separate components.

Although initialization code might be different for each module, there are some regular patterns that inform our signature design. In the main function of our fileserver in Figure 2 we require a store and a network handle, which correspond to the two arguments of the functor. This pattern is both common and expected: for functors, the initialization function typically requires the results of the initialization of each module arguments. This property holds in all the functors we have presented so far.

2.3 Reporting Errors

A modular system that allows for many implementations must also provide some mechanism for reporting errors. This error information must be simultaneously fine-grained enough for the developer to determine the appropriate recovery or failure mechanism, and coarse enough for different implementations to provide reasonable information in each possible failure case.

In MirageOS, we eschew the use of exceptions in favour of a more explicit approach using the standard `result` type and OCaml's *polymorphic variants* [Garrigue 2001]. The `result` type is a binary sum: a value of type `result` is either a "success" value `Ok v` or an "error" value `Error err`. Result also comes with monadic operations for chaining computations that can fail. OCaml's structurally typed polymorphic variants are distinguished by a leading backtick ` for each constructor: for instance, `` `Unknown_file s `` has the type ``[> `Unknown_file of string]``. Using structural typing makes it possible to combine multiple error types. For example, if `store_error` and `network_error` are polymorphic variant types, then `[> store_error | network_error]` denotes the combination: any value of either `store_error` or `network_error` is also a member of this type.

Figure 5 extends the `Store` signature to use these extensible error types. The revised `Store` signature exposes a type `error`, consisting of general errors expected to be encountered by any `Store` implementation, along with a pretty printer[Bonichon and Weis 2017] that builds a human-readable representation of an error (Line 3). By making the `error` type `private` [Garrigue 2006], we allow the implementation to provide a richer error type, as long as it contains at least the specified elements. Module type signatures with functions that may return an error use the `result` type to return either the result of a successful call or the relevant error information. For example, `Store` uses the `error` type together with `result` to provide structured error reporting for the `read` function (Line 7).

```
module type Store = sig
  type error = private [> ‘Unknown_file of string]
  val pp_error: Format.formatter -> error -> unit

  type t
  val read:
    t -> string -> (string, error) result
end
```

Fig. 5. `Store` extended with modular error handling.

```
1 module Store = Crunch
2 module Network = Http (TCPIP)
3 module MyServer = Server.Make (Store) (Network)
4
5 let () =
6   let crunch = CrunchModule.data in
7   let store = Crunch.create crunch in
8   let tcpip = TCPIP.create 80 in
9   let network = Network.create tcpip in
10  let server = MyServer.start store network in
11  run server
```

Fig. 6. Bringing it all together

This design has several appealing features. First, errors are extensible: individual implementations of `Store` can extend the `error` type with implementation-specific errors, Second, error checking is compositional: error types from multiple modules can be combined, and users can leverage the monadic API of the `result` type to chain computations. Third, error-checking is typed: OCaml's type system ensures that clients that abstract over `Store` signature can only match on errors (such as `Unknown_file`) exposed by the signature (although the pretty printer can always be called to log messages about other errors).

2.4 Gluing Modules Together

Section 2 described a modular file server and showcased several implementation for its sub-components. The flexibility of the modular approach allows us to assemble our application in a LEGO fashion by plugging modules together. Figure 6 combines the various components to create a self-contained file server that can be used in a POSIX environment. We use the `Crunch` module along with the `HTTP` functor applied to `TCPIP`. This results in two functor applications (Lines 2-3). We then need to initialize the various elements of our fileserver and launch it (Lines 9-10). Note how the initialisation code closely reflects the structure of the functor instantiation code, thanks to the regular pattern noted in Section 2.2.

Although it is straightforward, this code is not completely satisfactory to write by hand. Firstly, the code is repetitive: the structure of the functor applications and the state initialization is the same in each case (For example, the function applications in lines 9 and 10 mirror the functor applications in lines 2 and 3.) Furthermore, the code must be modified by hand each time we change a component of our application. (For example, using `Direct` in place of of `Crunch` would require changing both the functor applications and the initialization by hand.) Finally, while the code in this toy example is rather simple, its complexity rapidly increases in a realistic application. (For example, the unikernel that runs the MirageOS website contains more than 70 modules and a functor application depth of up to 10 for the devices it uses.) To handle such a rich ecosystem, we need better tooling.

3 FUNCTORIA: A TOOL TO GLUE MODULES AND SIGNATURES TOGETHER

Building executable applications from functor-heavy libraries involves significant boilerplate. OCaml's module language is much less flexible than its expression language: it does not support conditionals or more complex dependency requirements. This section presents a tool *functoria* and its DSL that acts as the glue language between the module and expression portions of the MirageOS application, allowing us to overcome these limitations.

The high-level goal of functoria is to automatically configure and build modular applications, such as the file server presented in Section 2, across the full variety of MirageOS backends. Functoria

```
type 'a typ
val (@->): 'a typ -> 'b typ -> ('a -> 'b) typ

type 'a impl
val ($): ('a -> 'b) impl -> 'a impl -> 'b impl

val foreign: -> string -> 'a typ -> 'a impl
```

Fig. 7. Library to describe modules and functors

```
type store
val store : store typ

type network
val network : network typ

val direct : string -> store impl
val crunch : string -> store impl
val netstore : (network -> store) impl
val tcpip : network impl
val http  : (network -> network) impl
```

Fig. 8. Devices combinators for the file server

```
let make_server =
  foreign
    "Server_modular.Make"
    (store @-> network @-> job)

let my_server =
  make_server $
  direct "data/" $
  (http $ tcpip)

let () = register "filesrv" [ my_server ]
```

Fig. 9. A `config.ml` file for the file server

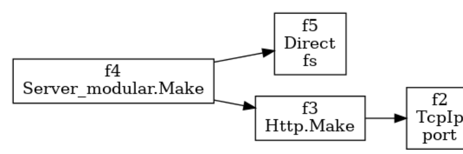

Fig. 10. Configuration graph for the file server

provides a CLI interface which takes arguments pertaining to the application to explicitly configure each of the constituent modules:

- `functoria configure --store direct --fs /my/files -p 42`
  configures the application to serve "/my/files" over a socket on port 42.
- `functoria configure --store crunch --fs /my/files -p 80`
  configures the application to create an HTTP file server serving the crunched files in "/my/files" on port 80. (The logic that interprets `80` to build an HTTP server is described below.)

The configuration process generates a file `main.ml` that applies all the application functors with concrete implementations, and also invokes the device initialisation code with the supplied configuration parameters. All the programmer has to do is to install any OCaml dependencies and invoke `make` to generate an executable unikernel from `main.ml`.

3.1 Configuring applications with functoria

Functoria relies on a configuration language that acts as a well-typed enforcer of the *structure* of the application (expressed by the programmer by functorising across its dependencies) and the *implementation* of those dependencies (expressed during the configuration process).

Figure 7 shows functoria's high-level operations for describing functors. A value of type `typ` represents a module type such as `Store` or `Network`. The `@->` operation builds a functor type from the types of its parameter and result: `store @-> network @-> job` represents the type of a functor that takes module arguments of type `store` and `network` and builds a module of type `job` representing the final unikernel. A value of type `impl` represents a module implementation. There is one operation, `$`, that corresponds to module application. The `foreign` function materializes a named module (i.e. creates a value of type `impl`) given its name and type.

Functoria also exposes particular values of type `typ` and `impl`, for the signatures and modules available in MirageOS (Figure 8). For example, `store` (of type `store typ`) corresponds to the `Store` signature (Figure 2), and `direct` (of type `store impl`) corresponds to the `Direct` implementation of type `Store` (Figure 3a). In each case the type index serves as a witness to ensure signature

```
module Key : sig
  type 'a k
  type +'a v
  val create: string -> 'a Arg.t -> 'a k
  val value: 'a k -> 'a v
  val pure: 'a -> 'a v
  val ($): ('a -> 'b) v -> 'a v -> 'b v
end

val if_: bool Key.v -> 'a impl -> 'a impl -> 'a impl
val match_: 'b Key.v -> ('b * 'a impl) list ->  'a impl
```

Fig. 11. Keys for parameterised applications.

```
let default_store dir : store impl =
  let i = Key.Arg.info
      ~doc:"Choose␣store" ["store"] in
  let arg = Key.Arg.(opt string "crunch"
      ~stage:‘Configure i) in
  let key = Key.create "store" arg in

  match_ (Key.value key) [
    "crunch", crunch dir;
    "direct", direct;
  ] ~default:(crunch dir)
```

Fig. 12. Using keys in a configuration pass

compatibility. These `typ` and `impl` values can be used for reflection (e.g. to list all the available implementations available for a given signature) as well as for composing functors to build devices.

Functoria also allows describing the various metadata associated with a module such as the packages it requires from OPAM (the OCaml package manager). Indeed, modules described by the configuration do not have to be immediately available in the current environment, but can be present in external libraries. The functoria tool will use OPAM to install all the required dependencies.

To configure a unikernel using these operations, the programmer creates a file `config.ml` that specifies how to combine the various module implementations. Figure 9 shows an example that corresponds to the handwritten code of Figure 6. The value `make_server` represents a functor `"Server_modular.Make"` with two parameters. The value `my_server` represents an application of that functor to two arguments: the module `direct`, and the result of applying the functor `http` to the server `tcpip`. Finally, the `register` function specifies and names the main module of the application.

Based on this code, Functoria will derive a graph that describes the structure of the application (Figure 10). This graph is used to synthethise everything related to the application: dependencies, initialisation and module code, documentation, package manager invocations, and so on.

3.2 Parametrized applications

The applications we have seen so far are very static: changing one of the modules requires rewriting either the code or the configuration. To provide the kind of flexibility needed in MirageOS applications, Functoria adds an additional ingredient: *keys*. The `Key` module (Figure 11) represents CLI arguments that can be used during configuration to determine which implementations to use in the generated code.

Figure 12 gives a new implementation of `Store` that supports selecting the storage mechanism at build time. The `default_store` value exposes the option `--store` to the command line and uses it to choose between the modules `Direct` or `Crunch`. The `Key.create` function declares a new key and the `Arg` module describes the CLI arguments (in this example, a simple enumeration). Finally, the `match_` function chooses an implementation based on the CLI key selection.

From a user perspective, this allows functoria to provide some extremely useful features for development. The user can choose between a filesystem or a built-in crunch store directly from the command line (e.g. by running `functoria configure --store crunch`). Functoria also generates the documentation of the application that describes all its keys, both as a Unix manual page and via the CLI:

```
$ functoria describe
Name       filesrv
Build-dir  .
Keys       store=crunch (default)
```

The `--store` key is only used during configuration; the `match_` combinator can only swap modules at configuration time. We use the `~stage:'Configure` argument to constrain this key to work at configure time. However, it is also possible to use keys dynamically at runtime. To demonstrate this, we can a new key to our file server to determine the port to listen to.

```
let port =
  let arg = Key.Arg.(opt int 80 (info ["p";"port"]))
  in Key.create "port" arg
```

We then use this key in the initialization code of the `TCPIP` module:

```
let network = TCPIP.create (Key_gen.port ())
```

The `--port` option can now be provided during both configuration and application startup. If the option is present during configuration, the value will be persisted and used as a default value during startup. MirageOS backends can supply more specific implementations for dynamic key lookup at runtime (for instance, via bootloader arguments, browser APIs, or conventional Unix environment variables).

In the examples so far, we have used keys in a "direct" manner: either by using their value directly for configuration (in the case of `--store`) or by passing the value off to the underlying application (for `--port`). We can also use keys for computations. For example, we define `default_network` which uses the `HTTP` functor if the port is 80 or 8080, but uses the normal `TCPIP` device otherwise. We use the fact that keys are split into two types: `Key.k`, which can be passed down to the runtime, and `Key.v`, which cannot be serialized but can be used in computations. We can use `Key.value` to obtain the value associated with `port`, and then apply `Key.pure` to our predicate to create a value that is not associated with a key. `$` allows us to apply the previous value to `port`. We can then use the resulting boolean value with `if_` to switch beetwen implementations.

```
let default_network : network impl =
  let is_http = Key.(pure (fun x -> x = 80 || x = 8080) $ value port) in
  if_ is_http (http $ tcpip) tcpip
```

`Key.value` equiped with `pure` and `$` (also often named `app`) forms an *applicative functor*[1]. The full library also provides other common applicative operators such as `map`.

3.3 Sharing and configuring devices

The `foreign` function is a specialised version of *configurable devices*. Configurable devices have an interface that describes the metadata provided by `foreign` modules (type, names, package descriptions and keys) and also the complete lifetime of a device: how to configure, build and detach it (Figure 13). The `connect` method specifies how to initialize the device—via a simple call to `start` in the case of `foreign` devices, but arbitrary initialization code in general for more complex cases. Once a configurable device has been defined, it can be encapsulated as an implementation via `impl`.

```
val impl: 'a configurable -> 'a impl

class type ['ty] configurable = object
  method ty: 'ty typ
  method name: string
  method module_name: string
  method keys: key list
  method connect:
    Info.t -> string ->
    string list -> string

  method packages: package list Key.value
  method configure: Info.t -> unit
  method build: Info.t -> unit
  method clean: Info.t -> unit
end
```

Fig. 13. API for configurable devices

The OCaml object system proves useful here. The definition of configurable devices using OCaml classes makes it possible to easily define classes of devices that are more specialised for a particular purpose. For example, we

[1] In the categorical sense. Not to be confused with ML functors!

could define a system where every device, once initialized, must add itself to a global list of devices. This can be encapsulated in functoria by providing a new function that generates the appropriate initialization code and used instead of `foreign`.

Various devices can sometimes have common dependencies. For example, a network device can be used both by HTTP devices and DHCP devices. However, it can't be assumed that devices are reentrant: many drivers for network connections should not initialize twice.

In functoria, devices are identified by both a module, which indicates their implementation, and a name, which defines their state. Functoria uses this name to decide which devices should be merged. If two devices have the same names, keys and–in the case of functors–are applied to the same arguments, they are considered equal. Equal devices share their state and their code. To force two devices to *not* be shared, it is sufficient to give them different names.

3.4 Building portable and flexible applications

We have made our example application more flexible than a typical monolithic Unix application, and are now able to change all the aspects of our file server simply by providing command line options. Our final configuration file, however, is barely more complex than it was at the beginning: Figure 14. Thanks to the interfaces provided by functoria, MirageOS implementors can provide combinators to make their devices easily usable in application configurations. The cost of this flexibility, of course, is a multiplication of command line options and devices. Functoria presents the configuration graph of the application in several formats to make it easier to reason about its modular structure. The graph for our final file server (Figure 15) shows configurable devices (rectangular nodes with a name and keys) and conditional configuration on keys (round nodes).

```
let make_server =
  foreign "Server_modular.Make"
    (store @->
     network @->
     job)

let my_server =
  make_server
  $ default_store "data/"
  $ default_network
```

Fig. 14. `config.ml` file for the file server application

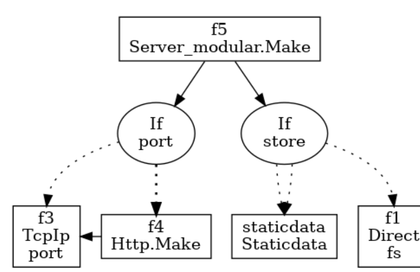

Fig. 15. Configuration graph of the file server.

4 DISCUSSION AND RELATED WORK

Growth of the OCaml libOS ecosystem. We have successfully used functoria as the core configuration language in MirageOS for the past three years. During that time it has scaled to manage the ever-expanding set of OS libraries written in pure OCaml to replace the original unsafe C versions. Functoria has been used to create many unikernel applications such as the self-hosted website whose configuration graph is rendered in Figure 1. The original vision of MirageOS was to provide a *complete* reimplementation of OS functionality in a type-safe language, and today the set of functoria module signatures in Table 1 show how far we have come in achieving this goal.

The `mirage` organisation on GitHub hosts over 100 repositories of independent OS libraries. MirageOS supports a variety of deployment targets and the examples in the "skeleton" repository compile to all of them. Some of the available MirageOS targets are:

- `unix`: maps filesystem and networking through to the Unix `libc` interfaces, resulting in a standard Unix application. This mode is useful during development of higher-level logic.

- xen: eliminates the dependency on a general-purpose OS and constructs a standalone kernel that boots on the Xen hypervisor. This requires a full device driver stack written in OCaml (from DHCP to TCP/IP to HTTP to TLS) that are all supported by functoria.
- hvt, virtio, muen and genode: these use the Solo5 hypervisor [Williams and Koller 2016] to run under KVM or directly on more specialised operating systems such as Meun or Genode. They also require a complete OCaml device stack instead of relying on an underlying OS.
- qubes: extends the Xen compilation target with extra device drivers to work on the QubesOS secure desktop Linux distribution, for example to firewall applications from each other.

There are also more experimental targets that link directly with embedded system bootlayers to run directly on open-source ARM or RISC-V hardware [Gala et al. 2016], providing a path to building highly secure and efficient IoT infrastructure. Note that all targets do not need to support all the possible device drivers—a Unix backend can only provide network sockets and not support direct Ethernet device signatures that are exposed by the Xen backend for example.

Module type	**Implementations**	**Comments**
Mirage_kv.RO	Crunch, Kv_Mem, Kv_unix, Mirage_tar, XenStore, Irmin, Filesystems	Read-only key-value stores allow to pass down immutable data to the unikernels such as web-pages, certificates, etc. Arbitrary filesystems can also be made into key-value stores.
Mirage_kv.RW	Wodan	Read-write key-value stores such as a pure OCaml store designed to run on SSDs.
Mirage_fs.S	Fat, Git, Fs_Mem, Fs_unix	Filesystem implementations.
Mirage_net.S	tuntap, vmnet, rawlink	Send and receive network packets.
ARP, IP, UDP, TCP	IPV4, IPV6, Qubesdb_IP, Udp, Updv4_socket, Tcp, Tcpv4_socket, …	Low-level implementations of Internet and Transport Protocols. Usually has two implementations: a complete reimplementation and one that delegates to the underlying OS.
STACK	Direct, Socket, Qubes, Static_IP, With_DHCP	Network stacks encapsulated for convenient usage. The stacks usually provide keys to customize its usage at configure and run time.
RANDOM	Stdlib, Nocrypto, Test	Random sources, either for normal or cryptographic purposes.
HTTP	Cohttp, Httpaf	HTTP servers implemented in term of an underlying STACK.
FLOW	Conduit.With_tcp, Conduit.With_tls	A generic abstraction for network flows that can be used with or without encryption.
DNS, DHCP, SYSLOG	Dns, Unix, Charrua_unix, Charrua, Syslog.Tcp, Syslog.Udp, Syslog.Tls	Protocols for various applications such as DNS, DHCP or Syslogs implemented in terms of an underlying STACK or FLOW.
	Jitsu, Irmin, …	High-level APIs that can provide extra functionalities. For instance, Jitsu [Madhavapeddy et al. 2015] can spawn new VMs on-demand.

Table 1. The MirageOS module ecosystem available on the opam package manager

Expressivity of Functoria. Our approach relies heavily on the OCaml module language to succeed, and functoria provides a partial embedding of the module system in the expression languages. Surprisingly, although modules have much more expressive type systems than our embedding supports, we found our subset sufficient for our organisational use.

Our observation is that when modules are used as a large scale organisation tool, it is generally to reduce the need for tightly coupled source codebases. This means converging towards a set of standardised signatures and avoiding subtyping hierarchies. The structural aspects of OCaml modules, while still useful, can then be emulated by nominal encodings and a use of phantom type parameters.

It is worth noting that OCaml significantly extends beyond the original roots of Standard ML. Features in OCaml such as applicative functors, Modula-2 style separate compilation and polymorphic variants have been essential when working with such a large number of modules. Examination of our use of these features in a large library such as our TCP/IP stack *vs* a more traditional ML implementation in the FoxNet [Biagioni et al. 2001] project is something we plan to examine to assess these extensions more closely.

Applicative vs. Generative. In OCaml, if module M is equal to N, the types provided by $F(M)$ and $F(N)$ are compatible. Such functors are called *applicative*[2] and are generally understood to be pure. Generative functors can have side-effects, and will thus generate fresh types when applied to their arguments. OCaml functors are applicative by default [Leroy 1994, 1995], but can be annotated to take on the generative behaviour. SML [MacQueen 1984] does not support applicative functors.

Since the functors we consider are pure (see Section 2.2), this applicative behaviour allows us to safely share codes across modules, including the result of functor applications. Devices are considered different only if their states or their dependencies are different, as shown in Section 3.3. Impure functors can nevertheless be useful. For instance, the MirageOS logging system relies on a generative functors to create a new logging interface per instantiation. Functoria supports impure functors by generating fresh device names for each functor application, which prevents sharing.

Alternative module languages. The constructs used in our approach can be found in module languages different from ML. Backpack [Kilpatrick et al. 2014] introduces a "linking calculus" for Haskell modules that supports features such as abstract signatures, separate compilation and sharing that are necessary for our approach. Scala's class calculus also supports a rich modularity toolset that covers most of our usecases via abstract classes and generics. MixML [Dreyer and Rossberg 2008] introduces structures that can be partially left abstract and filled later. This provides all the advantages of ML modules, including genericity, encapsulation and separate compilation but also support recursive modules which could be used for interdependent devices.

5 CONCLUSION

We have presented functor-driven development, an application architecture that leverages OCaml modules to structure application logic in a highly portable form that can be compiled across a variety of heterogenous targets.

Our implementation of the MirageOS unikernel framework has allowed us to successfully scale our ecosystem to hundreds of OCaml libraries. These libraries are packages which themselves contain thousands of OCaml modules. Overall we have millions of lines of modular and reusable OCaml code that provides clean-slate implementations of OS components – everything from device drivers to Internet protocols – that can be deployed on a large (and increasing) array of execution targets.

[2] Again, not to be confused with the categorical notion used for Keys in Section 3.2

REFERENCES

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.* 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

Edoardo Biagioni, Robert Harper, and Peter Lee. 2001. A Network Protocol Stack in Standard ML. *Higher-Order and Symbolic Computation* 14, 4 (01 Dec 2001), 309–356. https://doi.org/10.1023/A:1014403914699

R. Bonichon and P. Weis. 2017. Format unraveled. In *28. Journées francophones des langages applicatifs, Fréjus, France, January 7-7, 2017.*

Reto Buerki and Adrian-Ken Rueegsegger. 2013. Muen: An x86/64 separation kernel for high assurance. (2013). https://muen.sk/

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

Stephen Dolan and Mindy Preston. 2017. Testing with Crowbar. (2017).

Derek Dreyer and Andreas Rossberg. 2008. Mixin' up the ML module system. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 307–320. https://doi.org/10.1145/1411204.1411248

D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Copper Mountain, Colorado, USA, 251–266. https://doi.org/10.1145/224056.224076

N. Gala, A. Menon, R. Bodduna, G. S. Madhusudan, and V. Kamakoti. 2016. SHAKTI Processors: An Open-Source Hardware Initiative. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*. 7–8. https://doi.org/10.1109/VLSID.2016.130

Jacques Garrigue. 2001. Simple Type Inference for Structural Polymorphism. In *The Second Asian Workshop on Programming Languages and Systems, APLAS'01, Korea Advanced Institute of Science and Technology, Daejeon, Korea, December 17-18, 2001, Proceedings*. 329–343.

Jacques Garrigue. 2006. Private Row Types: Abstracting the Unnamed. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 44–60.

Thomas Gazagnaire and Vincent Hanquez. 2009. OXenstored: An Efficient Hierarchical and Transactional Database Using Functional Programming with Reference Cell Comparisons. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 203–214.

Scott Kilpatrick, Derek Dreyer, Simon L. Peyton Jones, and Simon Marlow. 2014. Backpack: retrofitting Haskell with interfaces. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 19–32. https://doi.org/10.1145/2535838.2535884

Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*.

Xavier Leroy. 1994. Manifest Types, Modules, and Separate Compilation. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 109–122. https://doi.org/10.1145/174675.176926

Xavier Leroy. 1995. Applicative Functors and Fully Transparent Higher-Order Modules. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 142–153. https://doi.org/10.1145/199448.199476

Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. 1996. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communications* 14, 7 (1996), 1280–1297.

David B. MacQueen. 1984. Modules for Standard ML. In *LISP and Functional Programming*. 198–207.

Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 559–573. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating*

Systems Principles (SOSP '17). ACM, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

David Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. 2010. Using Functional Programming Within an Industrial Product Group: Perspectives and Perceptions. *SIGPLAN Not.* 45, 9 (Sept. 2010), 87–92. https://doi.org/10.1145/1932681.1863557

Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016.*, Austin Clements and Tyson Condie (Eds.). USENIX Association.

M. Zalewski. 2014. (2014). http://lcamtuf.coredump.cx/afl/