Metaprogramming with ML modules in the MirageOS

Anil Madhavapeddy (speaker), Thomas Gazagnaire, David Scott, Richard Mortier

In this talk, we will go through how MirageOS lets the programmer build modular operating system components using a combination of functors and metaprogramming to ensure portability across both Unix and Xen, while preserving a usable developer workflow.

1 Introduction

MirageOS is a programming framework for constructing secure, high-performance network applications across a variety of cloud computing and mobile platforms. Code can be developed on a normal OS such as Linux or MacOS X, and then compiled into a fully-standalone, specialised unikernel that runs under the Xen hypervisor.[1] Since Xen powers most public cloud computing infrastructure such as Amazon EC2 or Rackspace, this lets servers run more cheaply, securely and with finer control than with a full software stack.

MirageOS uses the OCaml language, with libraries that provide networking, storage and concurrency support that work under Unix during development, but also Xen operating system drivers when being compiled for production deployment. The framework is fully event-driven using the Lwt library, with no preemptive threading.

Since the 2011 ML Workshop talk on MirageOS, it has undergone significant development and (since the December 2013 release) makes extensive use of the OCaml module system. There are now over 50 libraries that comprise the distribution, ranging from device drivers to network protocols to storage layouts.

2 Application Workflow

In this talk, we will go through how MirageOS lets the programmer build modular operating system components using a combination of functors and metaprogramming to ensure portability across both Unix and Xen, while preserving a usable developer workflow.

2.1 Device driver module types

Applications in MirageOS are functorized across all the operating system components that they depend on, with a library of module types provided to represent device driver functionality. For instance, below is the simplified type for a console output device.

```
module type DEVICE = sig
  type +'a io
  type t
  type error
  type id
  val id : t → id
  val connect: id →
    [`Error of error | `Ok of t] io
  val disconnect : t → unit io
end

module type CONSOLE = sig
  type error = [ `Invalid_console of string ]
  include DEVICE with type error := error
  val write : t → string → int → int → int
  val log_s : t → string → unit io
end
```

The application itself is built as a functor over these module types. Below is a simple console application that prints out a few hello world messages.

```
module Main (C: V1_LWT.CONSOLE) = struct
    let start c =
      let rec print () =
        C.log_s c "hello" >>= fun () →
        OS.Time.sleep 1.0 >>= fun () →
        C.log_s c "world" >>= print
      in print ()
    end
```

A more complex application, such as the MirageOS website[2] can be abstracted across many devices, for example with multiple storage nodes and network interfaces:

```
module Main
  (C: V1_LWT.CONSOLE) (FS: V1_LWT.KV_RO) (TMPL:
       V1_LWT.KV_RO)
  (S: Cohttp_lwt.Server)
  = struct ... end
```

The application above has a console, two key/value stores (one for files, and another for web templates), and an HTTP server.

2.2 Configuring an Application

The final stage is to actually produce an executable version of the application, which can take many forms depending on what the programmer is doing. For instance,

[1] See `queue.acm.org/detail.cfm?id=2566628`

[2] `http://openmirage.org/`

consider the following possible combinations for a typical application such as a website:

- A Unix binary serving HTTP traffic via kernel sockets (useful for development).
- A Unix binary serving HTTP via an OCaml network stack, using tuntap (useful during testing).
- A Xen unikernel that serves HTTP and drives virtual network ethernet device drivers (for production).

Each of these modes can be created by applying the application functor against the relevant modules that represent the right device drivers for that environment. This is automated via a command-line tool that parses an ML configuration file, and builds an appropriate `main.ml` file. Consider the trivial configuration for the hello world console example:

```
let main =
  foreign "Unikernel.Main" (console @→ job) in
register "console" [ main $ default_console ]
```

The goal of the configuration is to first specify the list of functors (i.e. the devices) that the application needs, and then to apply each of those functors to a concrete module implementation so that the application can actually run.

There are two interesting combinators provided here to do this. The `@->` combinator allows functor arguments to be appended, with the result satisfying a `job` module type that has the `start` and `stop` methods for controlling the application. Once the functor arguments have all been specified, it is applied to the `foreign` function to generate a job specifiction value.

The last step in configuration is to satisfy each of the functor arguments in the job with a concrete device driver. The `($)` combinator is used to apply real device drivers to `main` value, in this case simply the default console for either Unix or Xen.

The interface for these combinators is below:

```
type 'a typ
(** The type of values representing module
    types. *)

val (@→): 'a typ → 'b typ → ('a → 'b) typ
(** Construct a functor type from a type and an
    existing functor type. This corresponds to
    prepending a parameter to the list of
        functor
    parameters. *)

type 'a impl
(** The type of values representing module
    implementations. *)

val ($): ('a → 'b) impl → 'a impl → 'b impl
(** [m $ a] applies the functor [a] to the
    functor [m]. *)

val foreign: string → → 'a typ → 'a impl
(** [foreign name libs packs constr typ] states
     that the module named
    by [name] has the module type [typ]. *)
```

The implementation uses OCaml's GADTs to provide this functor construction interface:

```
type 'a foreign = {
  name: string;
  ty: 'a typ;
  libraries: string list;
  packages: string list;
}

type _ impl =
  | Impl: ('a, 'b) base → 'a impl
  (* base implementation *)
  | App: ('a, 'b) app → 'b impl
  (* functor application *)
  | Foreign: 'a foreign → 'a impl
  (* foreign functor implementation *)

and ('a, 'b) app = {
  f: ('a → 'b) impl; (* functor *)
  x: 'a impl;        (* parameter *)
}
```

The Mirage library provides all the device drivers as `impl` values, allowing the programmer to manipulate and assemble them conveniently from within a standard OCaml configuration script, avoiding the need for another configuration language.

2.3 Building the application

Once the application functor and configuration files are available, actually building the application is straightforward. The `mirage` command-line tool compiles the configuration script, dynamically links it in to access the result, and builds a `main.ml` file with all the relevant functor applications. This file is then compiled (with some additional options for the more special backends such as Xen) to deliver the final executable.

3 Discussion

The talk will cover the basics of this configuration combinator interface, and show how it is used in some fairly complex applications, such as a web server, a storage stack, and in different networking topologies. We are also currently developing new diverse backends to demonstrate the power of the functor approach, such as Raspberry Pi, Xen/ARM and JavaScript ports (which all run the same application source code).

More details and tutorials on MirageOS can be found at the homepage at `http://openmirage.org/`.