Genspio: Generating Shell Phrases In OCaml

Sebastien Mondet (**@smondet**)

Abstract

Genspio is a typed domain specific language embedded in OCaml that compiles terms to POSIX shell scripts or one-liners. It is used to build, for instance, complex deployment scripts which need to be run over SSH on hosts that may not have OCaml or any scripting language available.

The implementation is based on a GADT and has allowed us to scale to increasingly complex "Dev-Ops" deployments, thanks to the composability and modularity provided by OCaml itself. We describe these in Secotrec, for now our heaviest use of the Genspio EDSL.

While the released version has proved very useful, we detail quite a few very interesting future work problems.

Genesis

The project started as an experiment: "how far can we go like this?" and comes from two big sources of frustration. First, while running bioinformatics pipelines, we often need shell scripting to bypass idiosyncrasies of poorly written software. Second, in a "Dev-Ops" context, where we need to run increasingly complex deployment commands over SSH and through `sudo`; both have escaping rules that can be inconsistent even between minor versions. We want to manipulate a language with rudimentary but "proper" types, while making sure we rely only on the POSIX standard as common denominator.

Current Implementation

For the first version, we opted for a GADT-based internal representation of the typed AST as it is more practical to experiment with than, e.g., *typed-tagless-final* interpreters. In a future, more stable, version we may switch to a more modular representation.

We compile the EDSL to POSIX shell expressions; and we verify the implementation through quite thorough testing. Of course, there is nothing close to a formal specification of POSIX compatible shells, the tests try to verify the generated scripts against the informal documentation(s) of the standard.

The tests try to run on various shells found on the host (`bash`, `ksh`, `mksh`, `busybox`, etc.) and can be configured to also run on distant hosts with exotic operating systems over SSH. Hence one does not need to install any OCaml dependency to run the test suite.

From a user perspective, like any EDSL, the idea is to build values of type `'a Genspio.EDSL.t` through the combinators found in the `Genspio.EDSL` module; and compile them with functions from `Genspio.Compile`. See for instance:

```
let username_trimmed : string t =
  (* The usual shell-pipe operator is `||>`,
     `output_as_string` takes `stdout`
     from a `unit t` as a `string t`. *)
 (exec ["whoami"] ||> exec ["tr"; "-d"; "\\n"])
 |> output_as_string
```

The implementation of the compiler includes interesting encodings for portable shell *datatypes*, such as arbitrary strings (using octal representations) or `'a list` values (as multi-line strings). It also makes much easier the use of `trap` and Unix signals, as a method for jumping within a script. For instance one can error management similar to "poor-mans exceptions" with a high-level API:

```
with_failwith (fun error_function ->
  let get_user =
    (* the contents of `$USER`: *)
    getenv (string "USER") in
  (* The operator `=$=` is `string t` equality,
     it returns a `bool t` that
     we can use with `if_seq`: *)
  if_seq
    (get_user =$= username_trimmed)
     ~t:[ (* more commands *) ]
     ~e:[
       (* `$USER` is different from
          `whoami`, system is broken,
          we exit using the failwith
          funtion: *)
       error_function
         ~message:(string "I'm dying")
         ~return:(int 1)
     ])
```

The output, while difficult or even sometimes impossible to read by a human, is *meant to ensure* that the resulting script (or "one-liner") does not introduce portability errors on POSIX-compliant shells.

The EDSL provides also higher-level "library" constructs implemented with the EDSL, e.g. the function `EDSL.Command_line.parse` provides *typed* command line parsing through difference lists (similar to OCaml's `Printf` module).

More examples and documentation are available from the project page.

Scaling Deployments

The current version has already proved very useful, we provide our users with easy "dev-ops" deployments to setup and monitor themselves on cloud services (Google-Cloud and AWS), cf. Secotrec.

The deployments involve setting up Ketrew and Coclobas servers (released versions or arbitrary `opam` "pins") with their shared PostgreSQL database, and together with optional TLS tunneling, and an authentication proxy. Those are based on Docker-compose. Users can also setup Kubernetes clusters on the Google Container Engine or AWS-Batch compute environments. Other options include "preparing" pre-computed work environments for Biokepi workflows like Epidisco. Ketrew's "Getting Started" setup for beginners is also based to Secotrec-generated configurations.

Within this project, we have been able to *scale* far beyond we could do with ordinary shell scripts as the only common denominator, while preserving portability against, e.g., different versions of OpenSSH and `sudo`.

Embedding the language into OCaml allows us to express *generic and composable* shell "components," with a good separation of concerns thanks to the module system. For instance, we implement proper error management, with optional display and optional recovery/clean-up code.

As a symbolic milestone, by using `ssh -c …` with compiled Genspio one-liners, we quickly reached the maximal length a command line argument can have (130 KiB, see `xargs --show-limits`), and had to switch to generating shell scripts and using SCP to run them properly as `sh <script>`.

Related Work

The OCaml ecosystem has already been working around the untyped, dirty world of shell scripts. Examples include Jane Street's Shexp and Shcaml; although they do not solve the same problem as Genspio. Just like usual "system" libraries, such as Bos, they leverage OCaml to provide well designed APIs, but they require a special environment (or interpreter) on the host running the programs (i.e. they require OCaml dependencies).

Future Work

In version 0.0.0, Genspio deals with the following few basic types: `int`, `bool`, `string`, and `'a list`. Among them, we handle a single OCaml-like string type, and hence introduce a confusion that will need to be addressed. In the POSIX-shell world, just like in the C-language one, there are actually *two* conceptual string types:

- byte arrays (contents of files, standard output of commands, etc.) and
- NUL-terminated strings (shell variables or command line arguments cannot contain the `\000` character).

So far, Genspio can throw a compile-time exception when the compiler detects a misuse of NUL-terminated strings, but the check is obviously incomplete, and not based on proper types. We need to distinguish two string types at the EDSL level: e.g. `c_string` and `string`; and provide appropriate (potentially failing) conversion functions between them.

Other new data-types should also be integrated in the language:

- We need more expressive "streams" (i.e. iterable sequences) to handle big files or command outputs in a more flexible way.
- We need a notion of *embedded* shell scripts. Indeed, right now, the output of a compiler is sometimes used as a value in a higher-level EDSL term (e.g. as a command line argument); this is not composable and modular enough.

Another interesting direction is extending the compilers to non-POSIX interpreters, especially MS-Windows' Batch command language, as it is the most common non-POSIX environment.

To ease debugging, and for future integration in other layers of our workflows (cf. Ketrew), we also want a "display friendly" representation of EDSL terms, with smart logging functions which can allow the compiled shell scripts to reference code-locations in the pretty-printed output.

The Talk

The presentation will introduce the project while stressing on some details of the implementation with examples of use of the library.