agdARGS - Declarative Hierarchical Command Line Interfaces

Guillaume Allais
gallais@cs.ru.nl
Radboud University Nijmegen

1. Introduction

If functional programmers using statically typed languages (broadly) in the Hindley-Milner tradition have taken to boasting "If it typechecks, ship it!", the shared sentiment amongst the ones using dependently typed languages seemed for a long while to be closer to "Once it typechecks, shelve it!".

Over the years, there has been some outliers not only using Type Theory as a theorem proving tool but also demonstrating the practical benefits dependent types bring to the programmers' table. The nowadays classic definition of `printf` in a type-safe manner (Augustsson 1998) is a prime example. Other notable contributions in that vein include for instance parser DSLs and generators (Danielsson 2010; Stump 2016), and interactive systems (Brady 2014; Claret and Régis-Gianas 2015).

When writing an application in Type Theory, it is reasonable to expect the programmer to want to focus her attention on the core algorithms i.e. the parts that can be fully certified, dealing with sanitised data enriched with all sorts of fancy invariants. The wrapper code is not necessarily terribly exciting in comparison and tends to be treated more as an afterthought. Tanter and Tabareau (2015) have developed a nice library to facilitate the transition from weakly typed to strongly typed data whilst maintaining type safety. This potentially removes one layer of boilerplate.

Command line interfaces are another one of these layers of wrapper code. We offer a solution: a dependently typed DSL for defining declaratively hierarchical command line interfaces available at `https://github.com/gallais/agdARGS`.

2. Hierarchical Command Line Interfaces

A hierarchical command line interface is defined by:

- A **description** explaining the command's purpose. It has no influence on the implementation of the interface but is useful documentation for the end-user.
- A list of **subcommands**. They are themselves fully-fledged commands the user gets access to by mentioning a keyword. This makes it possible to give the interface a *hierarchical* structure. E.g. `git pull` accesses the subcommand `git-pull` from the main `git` interface with the keyword `pull`.
- A list of **modifiers** for the current command. They can be either **flags** one may set or **options** taking a parameter.
- Finally, strings which are neither subcommand keywords nor modifiers are considered **arguments** of the command.

With our library, the programmer can simply get an interface by specifying this structure. For instance[1], a command similar to UNIX's `wc` can be declared this way:

```
WordCount = record
{ description = "Print each file's counts"
; subcommands = noSubCommands
; arguments   = lotsOf filePath
; modifiers   =
  , "-l"       ::= flag "Newline count"
  ⟨ "-w"       ::= flag "Word count"
  ⟨ "-help"    ::= flag "Display help"
  ⟨ "-version" ::= flag "Version number" ⟨ ⟨⟩ }
```

Figure 1. The `wc` command's interface

3. Implementation Details

If the general structure of a command is set in stone, it cannot be the case for its subcommands and modifiers: they will vary from application to application in number and nature. This means that we need to design a first class representation of (extensible) records amenable to generic programming to deal with them.

3.1 Extensible Records

A record type is characterised by two things: a list of distinct field names and a type associated to each one of these fields. Given a decidable *strict* order on the type of names, we can make use of McBride's design principles (2014) to define a structure of lists sorted in strictly increasing order and thus only containing distinct elements. These will be our lists of names. The types associated to each one of these field names can be collected in a right-nested tuple computed by recursion on the list. A record value is then a right-nested tuple of values of the corresponding types.

Because there are so many computations at the type level, the unification machinery can get stuck on meta-variables introduced by implicit arguments. It is crucial for the usability of the library defining extensible records that some of these notions (the fields' types and the record value itself) are wrapped in an Agda record to guide the type inference.

The combinators `_::=_⟨_` and `⟨⟩` one can see in Figure 1 are also crucial to the library's usability: they make it possible to define the extensible record field by field without having to pay attention to the underlying representation where all the invariants are enforced.

[1] Unfortunately we lack the space necessary to give an example of an interface with subcommands

3.2 Commands as Rose Trees

The structure described in Section 2 is reminiscent of a rose tree and it is indeed implemented as one. It should now be folklore that rose trees benefit a lot from being defined as sized types (Abel 2010). It allows recursive traversals to weaponize higher-order functions without having to spend a lot of efforts appeasing the termination checker. An instance of such a higher order function we use to great effect is the fold over an extensible record of subcommands.

We made Size an implicit index so that it does not add any extra overhead from the programmer's point of view in places where it does not matter.

4. Generic Programming over Interfaces

The point of having a first order representation of Interfaces is, just like for any deeply-embedded DSL (Hudak 1996), to be able to write generic program against this representation.

4.1 Parsing

The most important use case is to harness the Interface declaration to make sense of the list of strings[2] passed to the executable called from the command line. The expected result of a successful parse is a path down the hierarchical structure of the interface selecting a subcommand together with a collection of recognized modifiers and arguments specific to that subcommand. Dependent types allow us to make this requirement explicit by indexing the path over the command it corresponds to. We write `ParsedCLI` c for the type of successful parses associated to the interface c. This parsing process can be decomposed in three successive phases:

1. The **subcommand selection** phase goes down the hierarchical interface picking subcommands based on the keywords provided by the user. As soon as a modifier or an argument for the current command in focus is found, the second phase starts.

2. The **modifier and arguments collection** phase now has settled for a given subcommand and tries to parse each new string as either one of its modifiers or, if that doesn't succeed, an argument.

3. At any point the string "--" can make the parser switch to the **argument collection** phase. It interprets each subsequent string as an argument to the command in focus. It is useful when arguments may look like modifiers e.g. `ls -l -- -l` lists (`ls`) in a long listing format (`-l`) the information about the file "-l" (`-- -l`).

We provide the user with a combinator readily putting various pieces together that should fit most use cases. Its takes an interface, a continuation for a successful parse and returns an `IO` computation:
`withCLI` $: \forall\, c\, (k : \texttt{ParsedCLI}\ c \to \texttt{IO}\ \top) \to \texttt{IO}\ \top$
Internally, `withCLI` performs a call to Haskell's `getArgs`, attempts to parse the list of strings it got back, and either prints the error to `stdout` if the parse failed or calls the continuation otherwise.

It is currently very simple but fits our need. We can imagine more elaborate variations on it. We could for instance "patch" on the fly the provided interface so that it supports all the common flags for requesting help (e.g. `-h`, `--help`, `-?`, etc.) and responds to them by displaying appropriate usage information.

4.2 Usage Information

It is indeed possible to exploit the available knowledge about the interface's hierarchical structure, the subcommands' names and their associated modifiers to generically produce usage information for the end-users' consumption. Our `usage` function traverses the interface tree in a depth-first manner: it starts by recursively displaying all the subcommands (if any) at an increased indentation level and then lists the modifiers for the current command. Ran on the `wc`-like interface described in Figure 1, it yields the output in Figure 2.

```
WordCount  Print each file's counts
  --help     Display help
  --version  Version Number
  -l         Newline count
  -w         Word count
```

Figure 2. Usage Information for the Interface in Fig. 1

5. Current Limitations and Future Work

Writing the continuation passed to `withCLI` can be a bit verbose when dealing with deeply nested interfaces. It ought to be possible to define combinators that make it easier to combine together small, self-contained subcommands each one handling its own branch of the subcommands tree.

It is rather common for interfaces to allow the grouping of flags which are one character long into compound flags (e.g. `tar -xz` is understood as `tar -x -z`) or to use the remainder of a one character long option as its parameter (e.g. `tar -xz -ffi` is understood as `tar -xz -f fi`). One can even mix the two e.g. `tar -xzffi`. The current parser does not handle these shortcuts.

The usage information is generated in a rather crude manner by putting raw strings together. A well-structured intermediate format describing in a simple manner the dependencies between blocks of text would be an ideal candidate for a refactoring. Wadler's Prettier Printer (2003) is a possible candidate.

Once the generation of usage information is well structured, it would be interesting to be able to generate proper `man` pages. An interesting problem to solve towards that goal is the generation of compact yet informative examples of valid usages.

References

A. Abel. Miniagda: Integrating sized and dependent types. In *Proceedings Workshop on Partiality and Recursion in ITP, PAR*, 2010.

L. Augustsson. Cayenne—a language with dependent types. In *International School on Advanced Functional Programming*, pages 240–267. Springer, 1998.

E. Brady. Resource-dependent algebraic effects. In *International Symposium on Trends in Functional Programming*, pages 18–33. Springer, 2014.

G. Claret and Y. Régis-Gianas. Mechanical verification of interactive programs specified by use cases. In *Proceedings of the Third FME Workshop on Formal Methods in Software Engineering*, pages 61–67. IEEE Press, 2015.

N. A. Danielsson. Total parser combinators. In *ACM Sigplan Notices*, volume 45, pages 285–296. ACM, 2010.

P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

C. T. McBride. How to keep your neighbours in order. In *ACM SIGPLAN Notices*, volume 49, pages 297–309. ACM, 2014.

A. Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016. ISBN 978-1-97000-127-3.

E. Tanter and N. Tabareau. Gradual certified programming in coq. In *Proceedings of the 11th Symposium on Dynamic Languages*, pages 26–40. ACM, 2015.

P. Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

[2] These are usually referred to as "command line arguments" e.g. in the specification of `getArgs` in the Haskell 98 report's "System Functions" section. We refrain from using that expression to avoid confusion with our Interface's notion of arguments