Bioinformatics, The Typed Tagless Final Way

Sebastien Mondet (**@smondet**)

Abstract

In the context of our Bioinformatics workflows library Biokepi, we provide an embedded domain specific language (EDSL) based on typed-tagless final interpreters. The first implementation was based on a Generalized Algebraic Data-Type (GADT) but the approach did not scale well enough, and most importantly, we really needed the EDSL to be extensible by the users of the library.

The idea is to use OCaml's module system instead of a GADT, to provide typed EDSLs as module types and write compilers as modules matching the signature; "programs" and program transformations are then functors taking such an implementation as argument.

This talk will present the new implementation of our EDSL, while discussing advantages and limitations of the new approach and trying to be a quick tutorial on type-tagless final interpreters in OCaml.

Brief History And Motivation

For the past 3.5 years, our research group has been developing a set of libraries and tools for running bioninformatics workflows. These tools include Ketrew, a workflow engine based-on an OCaml API providing expressive building blocks, called "workflow nodes," to construct complex computational pipelines. Ketrew is a very generic tool, used not only for bioinformatics but also for instance for system-administration tasks.

Biokepi is then a library of Ketrew workflow nodes which *wrap* bioinformatics tools: their installation, their use, and the fetching of their data requirements.

At first just as a nice experiment, we added a much higher-level API to Biokepi designed as a typed EDSL. The EDSL provided the level of granularity that a bioinformatician would want for designing a pipeline while hiding details that can be considered boilerplate. The implementation was based on a growing GADT definition whose terms were then compiled to more complex Ketrew workflows.

This EDSL worked out quite well for our users thanks to its conciseness and readability. It quickly grew to become our default API to describe workflows but we hit a couple of fundamental issues:

- The compiler (and the optimization passes) code grew to become difficult to manage: there were huge "match" statements, and the type errors were very unwelcoming to OCaml beginners.
- The EDSL was still missing basic functional programming constructs that limited its expressivity: proper `lambda`/`apply`, list functions, etc.
- More importantly, the EDSL could not be extended by *users* of the library: although there were a few "hooks" to achieve some tasks, the GADT-based definition is monolithic and *closed*.

We wanted what we already had plus, the *users* of the library to be able to:

- Extend the language to their specific needs.
- Re-use default compilers and transformation when implementing their extensions.
- Be able to implement transformations "by hand" i.e. sometimes it is easier and more readable to write the code than add an optimization pass.

We tried a few experiments that did not work out to full potential:

- Extensible types are a new exciting feature but they do not scale well for our needs: they loose a lot of the *"type-strength"* benefits and end-up not being extensible enough.
- Creating a basic functional language based-on GADTs with limited extensible bioinformatics terms.

Enter "Typed Tagless Final Interpreters" …

Extensible EDSLs

In July 2015, Oleg Kiselov introduced the Quel project to the OCaml mailing-list (later published at PEPM'16). The idea is to use OCaml's module system, to provide typed EDSLs as module types and write compilers as modules matching the signature; "programs" and program transformations are then functors taking such an implementation as argument. The approach ensures that the EDSL is fully extensible.

We reimplemented our EDSL following Kiselov's course notes (based on Haskell and type-classes) and the implementation of Quel. The new extensible EDSL has grown much bigger than the earlier implementation while keeping maintenance manageable.

The language is more powerful:

- It has more functional constructs: `lambda`/`apply`, list and pair functions, …
- It is easier to document thanks to being just module types.
- It is easier to maintain thanks to separation of concerns.
- And, most importantly, it is extensible by the users.

We maintain compilers to Ketrew workflows, JSON "pipeline descriptions," and Graphviz figures (`dot` language); as well as a simple program-transformation based on a more general and extensible "optimization framework." We have used the extensibility of the EDSL to provide application-specific language constructs, e.g. in the Epidisco project, which includes custom language constructs for data management and generating its HTML report.

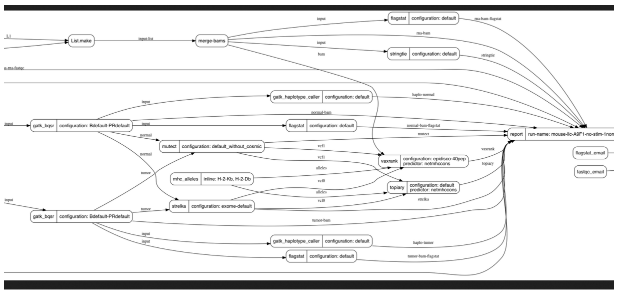

Figure 1: Example pipeline display from Epidisco

We have managed to get beginner OCaml users to add new features to the EDSL: while the initial setup can seem complex, EDSL modifications are mostly "compiler-error-guided" and the main difficulty is actually understanding the underlying bioinformatics tool and its idiosyncrasies.

Limitations

The approach has some caveats:

- Calling a compiler on a pipeline description requires 3 or 4 lines of functor-heavy code, which is slightly too much for a REPL/beginner-friendly experience.
- The "optimization framework" (a.k.a. extensible program transformations) still uses an intermediary GADT; this "cuts" the variance propagation so we cannot use subtyping in our definition of the EDSL semantics. We have not decided yet whether the mostly cosmetic optimization framework or the marginal usefulness of sub-typing in this context should be prioritized.

The Talk

The talk will walk through our implementation of Biokepi's EDSL while trying to be an introductory tutorial on module-based typed-tagless-final approaches.