Scrapping Your Dependently-Typed Boilerplate is Hard

Ahmad Salim Al-Sibahi
IT University of Copenhagen
Email: asal@itu.dk

I. Introduction

More complex programs that model programming languages or specific domain information end up with large nested data structures, and as a consequence changing even simple properties require a full traversal of the structure. The Haskell community has developed various techniques that try to reduce this boilerplate by providing generic frameworks querying and traversal, such as Scrap Your Boilerplate (SYB) [1] and Uniplate [2].

Since many of the modern dependently-typed languages such as Agda [3] or Idris [4] are inspired by Haskell, it would be reasonable to expect that such frameworks could be provided in these languages. However, as I will show in my presentation there are some unresolved issues that prevent these approaches from being used as-is.

II. Uniplate in a Minute

Uniplate provides a variety of functions, however the most interesting are `universeBi` and `transformBi` (see Figure 1). `universeBi` allows the programmer to extract all values of a target type from a source type and likewise `transformBi` allows the programmer to change all values of the target type in the source type.

```
universeBi :: Biplate from to ⇒ from → [to]
transformBi :: Biplate from to ⇒ (to → to) →
from → from
```

Fig. 1: Uniplate functions

Figure 2 shows an example model of blog posts, and presents two functions: `timestamps` and `capitaliseTitles`. `timestamps` uses generic querying to retrieve all values of type Timestamp, where `capitaliseTitles` applies the capitalisation function correctly to all target values of type `Title`.

III. Generic Programming in Dependent Types

Since SYB-style generic programming can reduce the amount of necessary yet tedious boilerplate that must be written, it seems desirable to also have such a capability in dependently-typed languages. In my presentation I will however highlight some of the difficulties I had encountered while trying to built such framework.

```
type Title = String
type Timestamp = Int
data Post = Single Title Timestamp | Aggregate [Post]

timestamps :: Post → [Timestamp]
timestamps = universeBi

capitaliseTitles :: Post → Post
capitaliseTitles = transformBi capitalise
```

Fig. 2: Sample Domain Model

A key reason why SYB-style generic programming is powerful is that the required methods can be generated automatically. However, in dependently typed languages this can have interesting consequences – even for monomorphicly restricted types – since indices might also be included in the resulting query or transformation. For example the type `Vec ℕ 2` has both a $\mathbb{N}$ index and $\mathbb{N}$ elements, and a naïvely generated query function will return the index in addition to the elements when retrieving all values of type `Nat`. More seriously, if a transformation can be performed on the index then this might result in an inconsistent result type, which means that the type checker must reject this code.

For polymorphic types, it can be hard to even specify how the query and transformation operators should work. For example it is not possible to return a value of just `Vec ℕ n` since `n` is unknown, and therefore the dependencies must be included as well such that the resulting type is $\Sigma_{n:\mathbb{N}}$ `Vec ℕ n`. Similarly, transformations must not only be able to reason about what dependencies are captured but must also be able to reason on the relationship between them in order to ensure that any changes made are still valid.

IV. Summary

In my presentation I will discuss the particular challenges that SYB-style generic programming presents to library implementer for dependently-typed languages, using examples motivated by practical programming. Additionally, a number of potential solutions will be considered, and the reason for their inadequacy will be discussed.

References

[1] R. Lämmel and S. P. Jones, "Scrap your boilerplate: a practical design pattern for generic programming," in *ACM SIGPLAN Notices*, vol. 38, no. 3. ACM, 2003, pp. 26–37.

[2] N. Mitchell and C. Runciman, “Uniform boilerplate and list processing,” in *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. ACM, 2007, pp. 49–60.

[3] U. Norell, “Dependently typed programming in agda,” in *Advanced Functional Programming*. Springer, 2009, pp. 230–266.

[4] E. BRADY, “Idris, a general-purpose dependently typed programming language: Design and implementation,” *Journal of Functional Programming*, vol. 23, pp. 552–593, 9 2013. [Online]. Available: http://journals.cambridge.org/article_S095679681300018X