Unified Type Checking for Type Classes and Type Families

Tom Schrijvers *
K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Martin Sulzmann
ITU, Denmark
martin.sulzmann@gmail.com

Keywords Haskell, type checking, type classes, type families

1. Introduction

Haksell has a rich type system with various complementary, interacting and overlapping features. In particular we think of the established *type classes*, with several extensions: multiple parameters, functional dependencies, ... In a recent proposal, a new feature is added to the Haskell language: type-level functions, or *type families* in GHC.

This multitude of type-level features is a blessing for programmers. A well-chosen combination of features allows the accurate expression of many problem domain semantics.

However, the plethora of features is also a nightmare for Haskell compiler writers, who have to implement and maintain all these features. For instance, GHC's core type checking modules for type classes and type families comprise approximately 3,1 kloC and 1,2 kloc, which is understood by very few people. Currently, no other Haskell system has managed to provide the same type class functionality as GHC.

The contributions of this work aim at recuding the implementation complexity of two Haskell type system features, type classes and type families:

- We reduce the type checking problem of type classes to a type checking problem of type functions.
- We propopse a small extension of the current type checking algorithm for type functions to cope with the above mapping.
- We sketch how to deal with evidence for both cases, dictionaries for type classes and coercions for type families.
- Our approach lifts a current restriction of type classes: it allows instance contexts to be extracted from dictionaries.

In the overview below, we illustrate the first two of these points.

2. Overview

The core of our idea is simple Functional Programming wisdom[1]:

> A predicate is (nothing more than) a function that returns a boolean value.

Hence, our idea is to represent a type class (a type-level predicate) as a type family (type-level function) returning a *type-level boolean*. These type-level booleans are represented by empty types:

```
data TRUE ; data FALSE
```

Consider for instance, the `Ord` type class with some instances (methods omitted):

```
class     Eq a          => Ord a
instance                   Ord ()
instance (Ord a, Ord b) => Ord (a,b)
```

These are mapped onto the following type families:

```
type family   OrdF a
type instance OrdF a = AND (EqF a) (OrdI a) -- (*)

type family   OrdI a
type instance OrdI ()    = TT
type instance OrdI (a,b) = AND (OrdF a) (OrdF b)
```

The encoding `OrdF a` reduces to `TT` iff `a` is instance of the `Ord` type class. If it is not, then `OrdF a` does not reduce to a value. The generic case (*) states that the class context must be satisfied and there must be an instance. The family `OrdI` captures the instances. These instances may recursively call `OrdF` to satisfy their contexts.

In order to verify whether a type class constraint $C\ \bar{t}$ holds, we verify whether its familiy encoding $CF\ t$ reduces to `TT`. In other words, we verify the equality constraint $CF\ t$ ~ `TT`. For this purpose, we can simply use the rewriting-based algorithm of type families (Schrijvers et al. 2008). It only needs one more equality rewriting rule to decompose the new `AND` symbol:

$$\texttt{AND}\ t_1\ t_2 \sim \texttt{TT} \quad \rightarrowtail \quad t_1 \sim \texttt{TT},\ t_2 \sim \texttt{TT}$$

Hence, we can make the following reduction for `Ord ((),())`:

```
OrdF ((),()) ~ TT
>-> AND (EqF ((),())) (OrdI ((),())) ~ TT
>-> EqF ((),()) ~ TT, OrdI ((),()) ~ TT
>->* AND (OrdF ()) (OrdF ()) ~ TT
>-> OrdF () ~ TT, OrdF () ~ TT
>->* TT ~ TT, TT ~ TT
```

3. Conclusion & Future Work

We have sketched a simplified approach to dealing with both type classes and type families with the same unified type checking algorithm. In future work we intend to elaborate and implement the approach, and prove it correct.

References

Tom Schrijvers, Simon Peyton-Jones, Manuel Chakravarty, and Martin Sulzmann. Type Checking with Open Type Functions. In *International Conference on Functional Programming*, 2008. Accepted.

* Post-doctoral researcher of the Fund for Scientific Research - Flanders.

[1] Often countered by the opposite Logic Programming wisdom.

[Copyright notice will appear here once 'preprint' option is removed.]