Restoring Confluence for Functional Dependencies via Type Families

Extended Abstract

Tom Schrijvers* and Martin Sulzmann

[1] Department of Computer Science, K.U.Leuven, Belgium
tom.schrijvers@cs.kuleuven.be
[2] School of Computing, National University of Singapore
sulzmann@comp.nus.edu.sg

Haskell-style functional dependencies [2] provide a relational specification of user-programmable type improvement [1] connected to type class instances. The more recent type families (also known as type functions) [3] equip the programmer with similar functionality, but in a functional form and decoupled from type classes. Functional dependencies are supported by both GHC and Hugs, while the most recent version of GHC also supports type functions.

There was an enthusiastic and lively debate about which feature should be adopted by the next Haskell standard, Haskell-Prime[3] Currently, further progress in the standardization appears to be stalled on this issue.

In this work, we attempt to rekindle the debate with new insights in type inference issues behind functional dependencies (FDs) and type functions (TFs), without taking sides. Specifically, we address the non-confluence issue of Constraint Handling Rules (CHRs) resulting from non-full FDs [4]. CHRs serve as a meta-language to describe the constraint solver underlying the type inferencer. Confluence of CHRs is important to ensure that the type inferencer is complete.

We propose an alternative encoding which achieves confluence even for non-full FDs. We achieve this result by encoding FDs via TFs. Technically, we could use a different CHR encoding but by using TFs to encode FDs we can also explain their commonalities and differences. We give a sketch of the alternative encoding below. The long version of this extended abstract will give an in-depth discussion of several encodings and also address issues when violating the (Weak) Coverage Condition [4].

The Non-Full FD Problem

Consider the following type class program:

```
class F a b c | a -> b
instance F a b Bool => F [a] [b] Bool
```

* Post-Doctoral Researcher of the Fund for Scientific Research - Flanders (Belgium) (F.W.O. - Vlaanderen).

[3] For example, see discussions on the Haskell-Prime and Haskell mailing lists.

The above FD is *non-full* because the type class parameter `c` plays no part in the FD `a -> b`. The unfortunate consequence of this non-fullness is that the CHRs resulting from the above program are non-confluent:

```
F a b1 c, F a b2 d ==> b1 = b2            (FD)
F [a] [b] Bool <=> F a b Bool           (Inst)
F [a] b c ==> b = [b1]                   (Imp)
```

These two distinct deriviations from the same set of constraints illustrate the non-confluence:

```
              F [a] [b] Bool, F [a] b2 d              (1)

>--> FD      F [a] [b] Bool, F [a] [b] d, b2 = [b]
>--> Inst    F a b Bool, F [a] [b] d, b2 = [b]       (2)

>--> Inst    F a b Bool, F [a] b2 d
>--> Imp     F a b Bool, F [a] [c] d, b2 = [c]       (3)
```

Both derivations yield a different result. We say that the CHRs are non-confluent which in terms of type inference means that we possibly lose completeness. Suppose that during type inference we try to verify that (2) is derivable from (1). However, we reduce (1) to (3). using the second derivation above. Constraints (2) and (3) differ therefore we (falsely) report that (2) is not derivable from (1).

Our Alternative FD Encoding

We claim that the above confluence is not inherent in the non-full FDs themselves, but an artefact of the CHR encoding of [4]. We suggest an alternative FD encoding that does not suffer from this confluence issue. This new encoding projects non-full FDs onto full FDs. We will use type function notation for full FDs. Thus showing that we can encode FDs via type functions.

The above type class program now translates to:[4]

```
type family FD a
type instance FD [a] = [FD a]

 class FD a ~ b => F a b c
 instance F a b Bool => F [a] [b] Bool
```

The point to note is that we project the non-full FD

```
class F a b c | a -> b
```

onto a "full" type function (type functions are always full by construction) and substitute the functional dependency `a -> b` by an equality constraint

```
type family FD a
class FD a ~ b => F a b c
```

[4] The symbol `~` is GHC's notation for type equality.

The important point is that the resulting TF program is confluent. Let us revisit the earlier example.

```
                F [a] [b] Bool, F [a] b2 d

>--> Imp    F [a] [b] Bool, F [a] b2 d, FD [a] ~ [b]
>--> Imp    F [a] [b] Bool, F [a] b2 d, FD [a] ~ [b], FD [a] ~ b2
>--> Subst  F [a] [b] Bool, F [a] b2 d, b2 ~ [b], FD [a] ~ b2
>--> Inst   F a b Bool, F [a] b2 d, b2 ~ [b], FD [a] ~ b2
>--> TF     F a b Bool, F [a] b2 d, b2 ~ [b], [FD a] ~ b2
>--> Subst  F a b Bool, F [a] b2 d, b2 ~ [b], [FD a] ~ [b]
>--> Decomp F a b Bool, F [a] b2 d, b2 ~ [b], FD a ~ b

>--> Inst   F a b Bool, F [a] b2 d
>--> Imp    F a b Bool, F [a] b2 d, FD [a] ~ b2
>--> TF     F a b Bool, F [a] b2 d, [FD a] ~ b2
>--> Imp    F a b Bool, F [a] b2 d, [FD a] ~ b2, FD a ~ b
>--> Subst  F a b Bool, F [a] [FD a] d, [FD a] ~ b2, FD a ~ b
>--> Subst  F a b Bool, F [a] [b] d, [b] ~ b2, FD a ~ b
```

Here `Subst` and `Decomp` are substitutions and decompositions of equations. The above are but two of the possible derivations, but one can verify that all derivations yield the same result.

Conclusion

In summary, we have shown that the non-confluence of non-full FDs is not inherent. With an alternative encoding in terms of TFs, confluence can be established. More generally, we believe that the design space for FD inference algorithms is all but exhausted, and that the TF and FD language features should not be judged by the adequateness of current inference algorithms.

In the extended version of this paper, we will explore two more encodings of non-full FDs and compare them to the above two. One replaces TFs by full FDs in the projection approach, and the other, proposed by Clause Reinke, avoids confluence issues via propagation CHRs.

References

1. M. P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1995.
2. M. P. Jones. Type classes with functional dependencies. In *Proc. of ESOP'00*, volume 1782 of *LNCS*. Springer-Verlag, 2000.
3. T. Schrijvers, M. Sulzmann, M. M. T. Chakravarty, and S. Peyton Jones. Towards open type functions for Haskell, 2007. Presented at IFL'07.
4. M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, 2007.