Elaboration on Functional Dependencies: Functional Dependencies Are Dead, Long Live Functional Dependencies!

Georgios Karachalias
KU Leuven
Belgium
georgios.karachalias@cs.kuleuven.be

Tom Schrijvers
KU Leuven
Belgium
tom.schrijvers@cs.kuleuven.be

Abstract

Functional dependencies are a popular extension to Haskell's type-class system because they provide fine-grained control over type inference, resolve ambiguities and even enable type-level computations.

Unfortunately, several aspects of Haskell's functional dependencies are ill-understood. In particular, the GHC compiler does not properly enforce the functional dependency property, and rejects well-typed programs because it does not know how to elaborate them into its core language, System F_C.

This paper presents a novel formalization of functional dependencies that addresses these issues: We explicitly capture the functional dependency property in the type system, in the form of explicit type equalities. We also provide a type inference algorithm and an accompanying elaboration strategy which allows all well-typed programs to be elaborated into System F_C.

CCS Concepts • **Theory of computation → Type structures**; • **Software and its engineering → Functional languages**;

Keywords Haskell, functional dependencies, System FC

ACM Reference Format:
Georgios Karachalias and Tom Schrijvers. 2017. Elaboration on Functional Dependencies: Functional Dependencies Are Dead, Long Live Functional Dependencies!. In *Proceedings of 10th ACM SIGPLAN International Haskell Symposium, Oxford, UK, September 7-8, 2017 (Haskell'17),* 15 pages.
https://doi.org/10.1145/3122955.3122966

1 Introduction

Type classes were originally introduced by Wadler and Blott [35] to make ad-hoc overloading less ad hoc. They first became highly successful in Haskell [20], were later adopted by other declarative languages like Mercury [10] and Coq [19], and finally influenced the design of similar features (e.g., concepts for C++ [8] and traits for Rust [3, 27]).

The feature was quickly and naturally generalized from single-parameter predicates over types to relations over multiple types. Unfortunately, these so-called *multi-parameter* type classes easily give rise to ambiguous situations where the combination of types in the relation can, as a matter of principle, not be uniquely determined. In many situations a functional relation between the types that inhabit a multi-parameter type class is intended. Hence, Jones proposed the *functional dependency* language extension [15], which specifies that one class parameter determines another.

Functional dependencies became quite popular, not only to resolve ambiguity, but also as a device for type-level computation, which was used to good effect, e.g., for operations on heterogeneous collections [18]. They were supported by Hugs, Mercury, Habit [11] and also GHC. However, the implementation in GHC has turned out to be problematic: As far as we know, it is not possible to elaborate all well-typed programs with functional dependencies into GHC's original typed intermediate language based on System F [7]. As a consequence, GHC rejects programs that are perfectly valid according to the theory of Sulzmann et al. [32]. What's more, GHC's type checker does accept programs that violate the functional dependency property.

With the advent of associated types [1] (a.k.a. type families) came a new means for type-level computation, with a functional notation. Because it too cannot be elaborated into System F, a new extended core calculus with type-equality coercions was developed, called System F_C [31]. However, it was never investigated whether functional dependencies would benefit from this more expressive core language. To date functional dependencies remain a widely popular, yet unreliably implemented feature. They are even gaining new relevance as functional dependency annotations on type families are being investigated [29].

Furthermore, as Jones and Diatchki [16] rightly pointed out, the interaction of functional dependencies with other features has not been formally studied. In fact, recent discussions in the Haskell community indicate an interest in the interaction of functional dependencies with type families (GHC feature request #11534). Moreover, the unresolved nature of the problem has ramifications beyond Haskell, as PureScript has also recently adopted functional dependencies.[1]

This paper revisits the issue of properly supporting functional dependencies, and provides a full formalization that covers an elaboration into System F_C for all well-typed programs.

Our specific contributions are:

- We present an overview of the shortcomings in the treatment of functional dependencies (Section 2).
- We provide a formalization of functional dependencies that exposes the implicit type-level function (Section 4).
- We present a type inference algorithm with evidence translation from source terms to System F_C that is faithful to the type system specification (Section 5).
- The meta-theory of our system states that the elaboration into System F_C is type-preserving (Section 6).

Haskell'17, September 7-8, 2017, Oxford, UK

ACM ISBN 978-1-4503-5182-9/17/09…$15.00
https://doi.org/10.1145/3122955.3122966

[1] http://goo.gl/V55whi