Elaboration on Functional Dependencies

Functional Dependencies Are Dead, Long Live Functional Dependencies!

Georgios Karachalias
Department of Computer Science
KU Leuven
Celestijnenlaan 200A
Leuven 3001, Belgium
georgios.karachalias@cs.kuleuven.be

Tom Schrijvers
Department of Computer Science
KU Leuven
Celestijnenlaan 200A
Leuven 3001, 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 F_C

ACM Reference format:
Georgios Karachalias and Tom Schrijvers. 2017. Elaboration on Functional Dependencies. In *Proceedings of ACM SIGPLAN Haskell Symposium 2017, Oxford, UK, September 7 – 8, 2017 (Haskell'17),* 15 pages.
DOI: 10.475/123_4

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).

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

2 Overview

2.1 Functional Dependencies

The concept of a *functional dependency* originates in relational database theory [28]: a relation R satisfies the functional dependency $X \to Y$, where X and Y are attributes of R, iff:

$$\forall (x, y_1), (x, y_2) \in R.\ y_1 = y_2 \qquad (1)$$

In other words, every X value in R is associated with precisely one Y value. The feature was first introduced in Haskell by Jones [15] as an extension to multi-parameter type classes and has been widely used over the years. The following variant of the well-known *collection* example [17] illustrates the feature:

$$\begin{array}{l}\textbf{class}\ \mathit{Coll}\ c\ e \mid c \to e\ \textbf{where} \\ \quad \mathit{singleton} :: e \to c\end{array}$$

The class *Coll* abstracts over collection types c with element type e. The functional dependency ($c \to e$) expresses that *"c uniquely determines e"*. Hence, functional dependencies have exactly the same meaning in Haskell as in relational database theory. After all, a multi-parameter type class like *Coll* can easily be seen as a relation over types. There is one main difference between Haskell type classes and database relations: The latter are typically defined extensionally (i.e., as a finite enumeration of tuples). In contrast, the former are given intensionally by means of type class instances (which can be seen as Horn clause rules) from which infinitely many tuples can be derived by means of type class resolution.

Besides supporting functional dependencies syntactically as documentation for the programmer, Haskell also supports functional dependencies semantically in two ways. Firstly, it enforces that the type class instances respect the functional dependency. This means for example that we cannot define two instances that associate different element types with the same collection type:

$$\begin{array}{l}\textbf{instance}\ \mathit{Coll}\ \mathit{Integer}\ \mathit{Bit}\ \ \ \textbf{where}\ \{\mathit{singleton}\ c = \ldots\} \\ \textbf{instance}\ \mathit{Coll}\ \mathit{Integer}\ \mathit{Byte}\ \textbf{where}\ \{\mathit{singleton}\ c = \ldots\}\end{array}$$

Secondly, functional dependencies give rise to more precise types and resolve ambiguities. For example, ignoring the functional dependency of *Coll*, function:

$$\mathit{singleton}_2\ c = \mathit{singleton}\ (\mathit{singleton}\ c)$$

has the ambiguous type:

$$\mathit{singleton}_2 :: (\mathit{Coll}\ c_1\ e, \mathit{Coll}\ c_2\ c_1) \Rightarrow e \to c_2$$

Type variable c_1 does not appear on the right of the $\Rightarrow$, which in turn means that no matter what argument we call *singleton*$_2$ on, c_1 will not be determined. Such ambiguous programs are typically rejected, since their runtime behavior is unpredictable (Section 6).

Yet, the functional dependency expresses that c_1 is not free, but uniquely determined by the choice of c_2, which will be fixed at call sites. Hence, if we take the functional dependency into account, *singleton*$_2$'s type is no longer ambiguous.

While functional dependencies are well-understood in the world of databases [28], their incarnation in Haskell is still surrounded by a number of major algorithmic challenges and open questions.

2.2 Challenge 1: Enforcing Functional Dependencies

Unfortunately, the current implementation of functional dependencies in the Glasgow Haskell Compiler does not enforce the functional dependency property (Equation 1) in all circumstances.[2] The reason is that no criteria have been identified to do so under the *Liberal Coverage Condition* [32, Def. 15], which regulates ways of defining functional dependencies indirectly through instance contexts. The following example illustrates the problem.

$$\textbf{class}\ C\ a\ b\ c \mid a \to b\ \textbf{where}\ \{\mathit{foo} :: a \to c \to b\}$$

$$\begin{array}{l}\textbf{class}\ D_1\ a\ b \mid a \to b\ \textbf{where}\ \{\mathit{bar} :: a \to b\} \\ \textbf{class}\ D_2\ a\ b \mid a \to b\ \textbf{where}\ \{\mathit{baz} :: a \to b\}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ D_1\ a\ b \Rightarrow C\ [a]\ [b]\ \mathit{Int}\ \ \ \textbf{where}\ \{\mathit{foo}\ [a]\ \_ = [\mathit{bar}\ a]\} \\ \textbf{instance}\ D_2\ a\ b \Rightarrow C\ [a]\ [b]\ \mathit{Bool}\ \textbf{where}\ \{\mathit{foo}\ [a]\ \_ = [\mathit{baz}\ a]\}\end{array}$$

$$\begin{array}{l}\textbf{instance}\ D_1\ \mathit{Int}\ \mathit{Int}\ \ \ \textbf{where}\ \{\mathit{bar} = \mathit{id}\} \\ \textbf{instance}\ D_2\ \mathit{Int}\ \mathit{Bool}\ \textbf{where}\ \{\mathit{baz} = \mathit{even}\}\end{array}$$

The above instances satisfy the Liberal Coverage Condition and imply that the 3-parameter type class C is inhabited by triples $([\mathit{Int}], [\mathit{Bool}], \mathit{Bool})$ and $([\mathit{Int}], [\mathit{Int}], \mathit{Int})$. If we project the triples on the functional dependency $a \to b$, then we see that $[\mathit{Int}]$ is associated with both $[\mathit{Int}]$ and $[\mathit{Bool}]$. In other words, the functional dependency is violated.

Yet, as the following two expressions show, GHC has no qualms about using both instances:

```
ghci> foo [1 :: Int] (True :: Bool)
[False]

ghci> foo [1 :: Int] (2 :: Int)
[1]
```

In short, GHC's current implementation of functional dependencies does not properly enforce the functional dependency property. This is not an implementation problem, but points at problem in the theory: it an open challenge how to do so under the Liberal Coverage Condition.

2.3 Challenge 2: Elaborating Functional Dependencies

GHC elaborates Haskell source programs into the typed intermediate language System F$_C$ [31], which is an extension of System F with type equality coercions. Among others, this elaboration process turns type class constraints into explicitly passed witnesses, the so-called type class dictionaries [9, 35].

Unfortunately, when it comes to functional dependencies, the elaboration process is incomplete: While Sulzmann et al. [32] provide the most concise and formal account of functional dependencies we are aware of, it has never been investigated how well-typed programs with respect to Sulzmann et al. [32] can be elaborated into System F$_C$.

Hence, GHC currently rejects those programs it cannot elaborate. It turns out, the problem is more general: due to the non-parametric semantics of functional dependencies, it is not possible to translate them to a statically-typed language like System F that features only parametric polymorphism. Indeed, as we discuss in Section 6.3, Hugs (which also translates to an intermediate language akin to System F) suffers from the same problem. Consider for instance the following program, which originates from GHC bug report #9627.

$$\begin{array}{ll}\textbf{class}\ C\ a\ b \mid a \to b & f :: C\ \mathit{Int}\ b \Rightarrow b \to \mathit{Bool} \\ \textbf{instance}\ C\ \mathit{Int}\ \mathit{Bool} & f\ x = x\end{array}$$

[2] See for example GHC bug reports #9210 and #10675.

This program is rejected because GHC has difficulty determining that type b equals *Bool* during the type-checking of function f. Yet, it is actually not difficult to see that the equality holds. From the functional dependency and the one instance for type class C, it follows that *Int* is uniquely associated with *Bool*. Hence, from the type class constraint C *Int* b, it must indeed follow that b equals *Bool*; b is not a type parameter that can be freely instantiated.

How to elaborate all well-typed Haskell programs with functional dependencies (with respect to the formal system of Sulzmann et al. [32]) into a typed intermediate language like System F_C is currently an open problem.

2.4 Challenge 3: Deduplicating Functional Dependencies

About ten years ago, a new type-level feature was introduced in Haskell that replicates much of the functionality of functional dependencies: (associated) type families [1]. They provide a functional, rather than a relational, notation for expressing a functional dependency between types. For instance, with associated type families we can express the *Coll* type class with a single parameter for the collection type and an associated *Elem* type family for the element type:

$$\begin{array}{l}
\textbf{class } \mathit{Coll}\ c\ \textbf{where} \\
\quad \textbf{type } \mathit{Elem}\ c :: * \\
\quad \mathit{singleton} :: \mathit{Elem}\ c \to c \\[4pt]
\mathit{singleton}_2 :: (\mathit{Coll}\ c, \mathit{Coll}\ (\mathit{Elem}\ c)) \Rightarrow \mathit{Elem}\ (\mathit{Elem}\ c) \to c \\
\mathit{singleton}_2\ c = \mathit{singleton}\ (\mathit{singleton}\ c)
\end{array}$$

This development means that GHC's Haskell dialect now supports two similar features. This is not necessarily problematic for modeling purposes, because each feature has its notational pros and cons. However, the separate support for both features gives rise to a lot of complexity in the type checker. While there has been a lot of speculation about the comparable expressive power of the two features, no formal comparison has been made. It is still an open engineering challenge to simplify the type checker by sharing the same infrastructure for both features.

2.5 Our Approach

The three challenges we have outlined above are all symptoms of a common problem: While we have a formalization of functional dependencies based on *Constraint Handling Rules* [32], we lack a formalization of functional dependencies that captures the functional dependency property properly within the type system and elaborates the feature into System F_C. The former provides a common ground for comparison with associated type families.

This paper provides such a formalization based on the conjecture of Schrijvers et al. [25] that functional dependencies can be translated into type families. In terms of the *Coll* example this idea means that we replace the functional dependency annotation $(c \to e)$ by a new type family *FD* and a "superclass" (see Section 3.2) constraint $(\mathit{FD}\ c \sim e)$ that captures the functional relation between the c and e parameters.

$$\begin{array}{l}
\textbf{class } \mathit{FD}\ c \sim e \Rightarrow \mathit{Coll}\ c\ e\ \textbf{where} \\
\quad \mathit{singleton} :: e \to c \\[4pt]
\textbf{type } \mathit{FD}\ c :: *
\end{array}$$

$$\begin{array}{llll}
\mathit{pgm} & ::= & \overline{\mathit{cls}}; \overline{\mathit{inst}}; \overline{\mathit{val}} & \textit{program} \\[6pt]
\mathit{cls} & ::= & \textbf{class } \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow \mathit{TC}\ \overline{a} \mid \overline{\mathit{fd}}^{m}\ \textbf{where } f :: \sigma & \textit{class} \\
\mathit{inst} & ::= & \textbf{instance } \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow \mathit{TC}\ \overline{u}\ \textbf{where } f = e & \textit{instance} \\
\mathit{fd} & ::= & a_1 \ldots a_n \to a_0 & \textit{fundep} \\
\mathit{val} & ::= & x = e & \textit{value binding} \\[6pt]
e & ::= & x \mid e_1\ e_2 \mid \lambda x.e \mid \textbf{let } x = e_1 \textbf{ in } e_2 & \textit{term} \\[6pt]
\sigma & ::= & \rho \mid \forall a.\sigma & \textit{type scheme} \\
\rho & ::= & \tau \mid Q \Rightarrow \rho & \textit{qualified type} \\
\tau & ::= & a \mid T \mid \tau_1\ \tau_2 \mid F(\overline{\tau}) & \textit{monotype} \\
u & ::= & a \mid T \mid u_1\ u_2 & \textit{type pattern} \\[6pt]
\phi & ::= & \tau \sim \tau & \textit{equality constraint} \\
\pi & ::= & \mathit{TC}\ \overline{\tau} & \textit{class constraint} \\
Q & ::= & \phi \mid \pi & \textit{type constraint} \\
C & ::= & \epsilon \mid C, Q & \textit{type constraint set} \\[6pt]
S & ::= & \forall \overline{a}.C \Rightarrow Q & \textit{constraint scheme}
\end{array}$$

Figure 1. Source Syntax

Moreover, we derive an appropriate *FD* instance for every *Coll* instance. For example, the list instance:

$$\begin{array}{l}
\textbf{instance } \mathit{Coll}\ [e]\ e\ \textbf{where} \\
\quad \mathit{singleton}\ x = [x]
\end{array}$$

gives rise to the type family instance:

$$\textbf{type } \mathit{FD}\ [e] = e$$

Intuitively, this transformation implements an alternative definition of a functional dependency: A relation R satisfies the functional dependency $X \to Y$, where X and Y are attributes of R, iff

$$\exists f : X \to Y.\ \forall (x, y) \in R.\ f(x) = y \qquad (2)$$

This paper addresses the challenges of functional dependencies with a formalization of the above idea in terms of a fully formal elaboration into System F_C. Our elaboration represents type class dictionaries with GADTs that hold evidence for the functional dependencies. Unlike for other dictionary fields, pattern matching to extract this evidence cannot be encapsulated in projection functions but has to happen at use sites. While GHC already uses this approach in practice for equalities in class contexts, as far as we know this approach has never been formalised before.

3 Logical Reading of FDs and Type Classes

Before presenting our formalization of functional dependencies in the next section, this section revisits the logical reading of type classes and functional dependencies.

To aid readability, we first present the source syntax in Section 3.1, and defer the logical interpretation of the class system to Sections 3.2 and 3.3.

3.1 Syntax

The syntax of source programs is given in Figure 1. A program *pgm* consists of class declarations $\overline{\mathit{cls}}$, instance declarations $\overline{\mathit{inst}}$ and variable bindings $\overline{\mathit{val}}$.

The syntax of class declarations and instances is standard; classes and instances are allowed to have multiple type arguments, and class declarations can also be annotated with functional dependencies. Functional dependencies take the simple form $a_1 \ldots a_n \rightarrow a_0$.[3] We also explicitly separate the type variables $\overline{a}$ that appear in the class/instance head from type variables $\overline{b}$ that appear only in the class/instance context $\overline{\pi}$.

Expressions comprise a λ-calculus, extended with let bindings.

The syntax of types also appears in Figure 1; like all HM(X)-based type systems, we discriminate between monotypes τ,[4] qualified types ρ and polytypes σ. Finally, we denote (possibly non-linear) type patterns as u.

Note that monotypes τ include type family applications $F(\overline{\tau})$, in addition to the standard forms. Although we disallow type families in the source text, as we illustrate in the rest of the section, in our formalization each functional dependency gives rise to a type family declaration.

In order to reduce the notational burden, we omit all mention of kinds and assume that each class has exactly one method.

3.2 Logical Reading of Class Declarations

A class declaration of the form

$$\textbf{class } \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{a} \mid fd_1, \ldots, fd_m \textbf{ where } f :: \sigma$$

gives rise to two kinds of constraint schemes:

Superclass Constraint Schemes

$$S_{C_\pi} = \forall \overline{a}.TC\ \overline{a} \Rightarrow \theta(\pi) \quad \forall \pi \qquad \text{(CS1a)}$$

where $\theta = det(\overline{a}, \overline{\pi})$ (we explain the meaning of function det below). This constraint scheme expresses the logical reading of the superclass relation: Given a class constraint, we can derive that each of the superclass constraints is also satisfied. A simple example which illustrates this is the following:

$$\begin{array}{ll} \textbf{class } Eq\ a \textbf{ where} & \textbf{class } Eq\ a \Rightarrow Ord\ a \textbf{ where} \\ \quad eq :: a \rightarrow a \rightarrow Bool & \quad ge :: a \rightarrow a \rightarrow Bool \end{array}$$

The Ord class gives rise to the superclass constraint scheme:

$$\forall a.\ Ord\ a \Rightarrow Eq\ a$$

(Observe that the implication arrow points in the opposite direction of the one in the class declaration!) As a consequence, we do not have to mention the $Eq\ a$ constraint explicitly in the signature of the function gt below. Instead, it can be derived implicitly by the type-checker from the given $Ord\ a$ constraint by means of the scheme.

$$\begin{array}{l} gt :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool \\ gt\ x\ y = ge\ x\ y\ \wedge\ not\ (eq\ x\ y) \end{array}$$

Functional dependencies complicate matters. Consider deriving the superclass scheme for class D:

$$\begin{array}{l} \textbf{class } C\ a\ b \mid a \rightarrow b \\ \textbf{class } C\ a\ b \Rightarrow D\ a \end{array}$$

By simply selecting the corresponding type in the class context, we get the following, broken constraint scheme

$$\forall a.D\ a \Rightarrow C\ a\ b$$

where b is free! The source of this problem is that b is actually *existentially quantified*, a more appropriate formulation would be:

$$\forall a.D\ a \Rightarrow \exists! b.C\ a\ b$$

Our language of constraint schemes, which reflects Haskell's type system, does not support top-level existentials though, so such an implication is not directly expressible within the language. Yet, there is a way to express b in terms of the in-scope variable a: Given that class C comes with a functional dependency, there *exists* a function symbol F_C such that $F_C\ a \sim b$ (according to Equation 2).[5] Hence, we can substitute b with $F_C\ a$ in the above broken scheme to obtain the valid:

$$\forall a.D\ a \Rightarrow C\ a\ (F_C\ a)$$

The computation of such a substitution is performed by function det, the formal description of which we defer until Section 4.1.4.

This example makes apparent why such class declarations have been rejected by GHC until now: Without a way of explicitly expressing b in terms of a, there is no way to express this relation within the type system.

Functional Dependency Constraint Schemes Every functional dependency $fd_i \equiv a_{i_1} \ldots a_{i_n} \rightarrow a_{i_0}$ that accompanies the class logically corresponds to the following constraint scheme:

$$S_{C_{fd_i}} = \forall \overline{a}.\ TC\ \overline{a} \Rightarrow F_{TC_i}\ a_{i_1}\ \ldots\ a_{i_n} \sim a_{i_0} \qquad \text{(CS1b)}$$

This constraint scheme directly expresses the functional dependency: Given $TC\ \overline{a}$, we know that *there exists* a function f such that $a_{i_0} = f(a_{i_1}, \ldots, a_{i_n})$. We explicitly give this type-level function for the i-th functional dependency of class TC the name[6] F_{TC_i}. For example, our running example $Coll$ gives rise to one such functional dependency constraint scheme:

$$\forall c\ e.Coll\ c\ e \Rightarrow F_{Coll_1}\ c \sim e$$

Notice how this scheme realizes the first part of the informal transformation of Schrijvers et al. [25]: If we (notionally) replace the functional dependency with a superclass equality constraint, then Scheme CS1b is just a special case of Scheme CS1a.

3.3 Logical Reading of Class Instances

A class instance

$$\textbf{instance } \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u} \textbf{ where } f = e$$

also yields two kinds of constraint schemes:

Instance Constraint Scheme

$$S_{I_{\overline{\pi}}} = \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u} \qquad \text{(CS2a)}$$

This constraint scheme directly expresses the logical reading of the class instance: *"If the context $\overline{\pi}$ is satisfied, then $(TC\ \overline{u})$ also holds"*. For example, the list instance of Eq yields the scheme:

$$\forall e.Eq\ e \Rightarrow Eq\ [e]$$

[3]We do not consider *multi-range FDs* [32] which can be desugared into simple functional dependencies [28].
[4]Arrow types $(\tau_1 \rightarrow \tau_2)$ are expressed as $((\rightarrow)\ \tau_1)\ \tau_2$.

[5]We use symbol "$\sim$" to denote type equality, following the convention of earlier work on System F_C [31].
[6]As a matter of fact, Jones suggested assigning names to functional dependencies as an interesting extension in his original work [15].

FD Witness Constraint Schemes Every functional dependency $fd_i \equiv a_{i_1} \dots a_{i_n} \rightarrow s_{i_0}$ that accompanies the class also yields a constraint scheme for the instance:

$$S_{I_{fd_i}} = \forall \overline{c}_i.\, F_{TC_i}\ u_{i_1}\ \dots\ u_{i_n} \sim \theta(u_{i_0}) \tag{CS2b}$$

where $\overline{c}_i = fv(u_{i_1}, \dots, u_{i_n})$ and $\theta = det(\overline{c}_i, \overline{\pi})$. The idea of this constraint scheme is to (partly) define the function F_{TC_i} that witnesses the functional dependency. The partial definition covers the subset of the domain that is covered by the class instance. Other instances give rise to schemes that cover other parts of the function. For example, the following program:

$$\begin{array}{l}\textbf{class}\ TC\ a\ b \mid a \rightarrow b\\ \textbf{instance}\ TC\ Int\ Bool\\ \textbf{instance}\ TC\ (Maybe\ a)\ a\end{array}$$

gives rise to the following FD witness constraint schemes (axioms):

$$\begin{array}{ll}F_{TC}\ Int & \sim Bool\\ \forall a.F_{TC}\ (Maybe\ a) & \sim a\end{array}$$

Observe that these schemes are essentially type family instances.

Similarly to the superclass constraint schemes, FD witness constraint schemes are quite challenging to derive in the general case. At first sight, it seems easy to determine the right-hand side u_{i_0}: simply take the corresponding parameter in the instance head. This indeed works for the simple examples above, but fails in more advanced cases.

Consider deriving the scheme for the following instance:

$$\textbf{instance}\ TC\ a\ b \Rightarrow TC\ [a]\ [b]$$

Simply selecting the corresponding type in the instance head, gives:

$$S_{I''_{fd}} = \forall a.F_{TC}\ [a] \sim [b]$$

This equation is broken again, as type variable b is free.

What happens here is that $[b]$ is not determined directly by the other instance argument $[a]$, but indirectly through the instance context ($TC\ a\ b$). If we apply the FD Constraint scheme to this instance context, we obtain that ($F_{TC}\ a \sim b$). This equation allows us to express b in terms of a. If we substitute it into the broken equation above, we do obtain a valid defining equation.

$$S_{I''_{fd}} = \forall a.F_{TC}\ [a] \sim [F_{TC}\ a]$$

Essentially, the FD witness constraint scheme realizes the second part of the transformation of Schrijvers et al. [25]: For every class instance, we generate a new type family instance for each functional dependency of the class.

In general, the derivation of a proper defining equation may require an arbitrary number of such substitution steps. We return to this in Section 4.1.4.

4 Type Checking

We now turn to the declarative type system of Haskell with functional dependencies. Our formalization utilizes the syntax we presented in Section 3.1, which we now augment with the typing and instance environments:

$$\begin{array}{lcll}I & ::= & \bullet \mid I, S & \textit{instance environment}\\ \Gamma & ::= & \bullet \mid \Gamma, a \mid \Gamma, x : \sigma & \textit{typing environment}\end{array}$$

The instance environment I is prepopulated by the constraint schemes induced by the program's class and instance declarations, and is then extended with *local assumptions* when moving under a qualified type. The typing environment Γ is standard, but we omit kind information for brevity.

4.1 The Type System

Figure 2 presents the typing rules for our system. By design, it closely resembles the system of Chakravarty et al. [1].

Similarly to earlier work on type class elaboration [9] and associated types [1, 2], we maintain the constraint schemes (context reduction rules) as part of the instance environment I.

4.1.1 Type Checking Terms

The judgment for typing terms is presented in Figure 2 and takes the form $I; \Gamma \vdash_{\mathsf{tm}} e : \sigma$. Most of the rules correspond to those for the polymorphic lambda calculus [7] with qualified types [12]. The only interesting case is Rule TmCast, which allows for casting the type of a term e from σ_1 to σ_2, as long as the equality of these types can be established. The satisfiability of the equality constraint is established via the *constraint entailment relation* $I \models S$, which is the focus of the next subsection.

Furthermore, Rules ($\forall E$) and ($\rightarrow I$) check the well-formedness of types via relation $\Gamma \vdash_{\mathsf{ty}} \sigma$. Since it is entirely standard, we omit its definition from our main presentation (it can be found in technical Appendix A).

4.1.2 Constraint Entailment

The *constraint entailment relation* takes the form $I \models S$ and is given by the following rules:

$$\frac{I \models \forall a.S}{I \models [\tau/a]S}\ \textsc{Inst} \qquad \frac{}{I \models \tau \sim \tau}\ \textsc{Refl} \qquad \frac{I \models \tau_2 \sim \tau_1}{I \models \tau_1 \sim \tau_2}\ \textsc{Sym}$$

$$\frac{I \models \tau_1 \sim \tau_2 \qquad I \models \tau_2 \sim \tau_3}{I \models \tau_1 \sim \tau_3}\ \textsc{Trans} \qquad \frac{I \models Q \Rightarrow S \qquad I \models Q}{I \models S}\ \text{MP}$$

$$\frac{S \in I}{I \models S}\ \textsc{Spec} \qquad \frac{I \models [\tau_1/a]Q \qquad I \models \tau_1 \sim \tau_2}{I \models [\tau_2/a]Q}\ \textsc{Subst}$$

Our system needs to check entailment of both type class and equality constraints, which is reflected in its rules: Rules Refl, Trans, Sym and Subst constitute the four standard equality axioms. MP is the elimination rule, and Rule Inst instantiates a constraint scheme with a monotype. Rule Spec is the standard axiom rule. Like in Jones' *Constructor Classes* [13], the entailment relation $I \models S$ is transitive, closed under substitution and monotonic (if $I_1 \models S$ then $I_1, I_2 \models S$).

4.1.3 Type Checking Declarations

Declaration typing also appears in Figure 2 and is –for the most part– standard. Since the value binding typing relation ($I; \Gamma_1 \vdash_{\mathsf{val}} val : \Gamma_2$) and the program typing relation ($\vdash_{\mathsf{pgm}} pgm$) are uninteresting, we elide them from our presentation (they can be found in technical Appendix A). Typing Rules Class and Instance type check class and instance declarations, respectively, and give rise to the constraint schemes we presented in Section 3. Both rules differ from earlier work in two ways:

1. The *Liberal Coverage Condition* [32, Def. 15] is enforced by the specification ($fv(\theta_i(u_{i_0})) \subseteq fv(\overline{u}^{i_n}), \theta_i = det(fv(\overline{u}^{i_n}), \overline{\pi})$), rather than being an additional, external restriction. This design choice is rather easy to motivate: If the domain of the functional dependency does not determine (even indirectly, via the context)

$\boxed{I;\Gamma \vdash_{\mathsf{tm}} e : \sigma}$ Term Typing

$$\frac{(x : \sigma) \in \Gamma}{I;\Gamma \vdash_{\mathsf{tm}} x : \sigma}\ \textsc{TmVar} \qquad \frac{I;\Gamma \vdash_{\mathsf{tm}} e : \sigma_1 \qquad I \models \sigma_1 \sim \sigma_2}{I;\Gamma \vdash_{\mathsf{tm}} e : \sigma_2}\ \textsc{TmCast} \qquad \frac{I;\Gamma, a \vdash_{\mathsf{tm}} e : \sigma \qquad a \notin \Gamma}{I;\Gamma \vdash_{\mathsf{tm}} e : \forall a.\sigma}\ (\forall I) \qquad \frac{I;\Gamma \vdash_{\mathsf{tm}} e : \forall a.\sigma \qquad \Gamma \vdash_{\mathsf{ty}} \tau}{I;\Gamma \vdash_{\mathsf{tm}} e : [\tau/a]\sigma}\ (\forall E)$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash_{\mathsf{ty}} \tau_1 \qquad I;\Gamma, x : \tau_1 \vdash_{\mathsf{tm}} e : \tau_2}{I;\Gamma \vdash_{\mathsf{tm}} \lambda x.e : \tau_1 \to \tau_2}\ (\to I) \qquad \frac{I;\Gamma \vdash_{\mathsf{tm}} e_1 : \tau_2 \to \tau_1 \qquad I;\Gamma \vdash_{\mathsf{tm}} e_2 : \tau_2}{I;\Gamma \vdash_{\mathsf{tm}} e_1\ e_2 : \tau_1}\ (\to E)$$

$$\frac{I, Q;\Gamma \vdash_{\mathsf{tm}} e : \rho}{I;\Gamma \vdash_{\mathsf{tm}} e : Q \Rightarrow \rho}\ (\Rightarrow I) \qquad \frac{I;\Gamma \vdash_{\mathsf{tm}} e : Q \Rightarrow \rho \qquad I \models Q}{I;\Gamma \vdash_{\mathsf{tm}} e : \rho}\ (\Rightarrow E) \qquad \frac{I;\Gamma, x : \tau \vdash_{\mathsf{tm}} e_1 : \tau \qquad x \notin dom(\Gamma) \qquad I;\Gamma, x : \tau \vdash_{\mathsf{tm}} e_2 : \sigma}{I;\Gamma \vdash_{\mathsf{tm}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma}\ \textsc{TmLet}$$

$\boxed{I;\Gamma \vdash_{\mathsf{cls}} cls : I_c;\Gamma_c}$ Class Declaration Typing

$$\frac{\theta = det(\overline{a},\overline{\pi}) \qquad unambig(\overline{b},\overline{a},\overline{\pi}) \qquad \Gamma,\overline{a} \vdash_{\mathsf{ty}} \sigma \qquad fd_i \equiv \overline{a}^{i_n} \to a_{i_0}}{I;\Gamma \vdash_{\mathsf{cls}} \mathbf{class}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{a} \mid \overline{fd}^{m}\ \mathbf{where}\ f :: \sigma : [\overline{\forall \overline{a}.TC\ \overline{a} \Rightarrow \theta(\pi)}, \overline{\forall \overline{a}.TC\ \overline{a} \Rightarrow F_{TC_i}(\overline{a}^{i_n}) \sim a_{i_0}}^{m}];[f : \forall \overline{a}.TC\ \overline{a} \Rightarrow \sigma]}\ \textsc{Class}$$

$\boxed{I_c;I_i;\Gamma \vdash_{\mathsf{inst}} inst : I}$ Instance Declaration Typing

$$\frac{\begin{array}{c}\theta = det(\overline{a},\overline{\pi}) \qquad unambig(\overline{b},\overline{a},\overline{\pi}) \qquad (f : \forall \overline{a}.TC\ \overline{a} \Rightarrow \sigma) \in \Gamma \qquad I_c, I_i, \overline{\pi};\Gamma \vdash_{\mathsf{tm}} e : [\overline{u}/\overline{a}]\sigma \\ \theta_i = det(fv(\overline{u}^{i_n}),\overline{\pi}) \qquad fv(\theta_i(u_{i_0})) \subseteq fv(\overline{u}^{i_n}) \qquad \forall(fd_i \equiv \overline{a}^{i_n} \to a_{i_0}) \qquad I_c, I_i, [\overline{u}/\overline{b'}]\overline{\pi} \models [\overline{u}/\overline{b'}]Q \qquad \forall(\forall \overline{b}'.TC\ \overline{b}' \Rightarrow Q) \in I_c\end{array}}{I_c;I_i;\Gamma \vdash_{\mathsf{inst}} \mathbf{instance}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u}\ \mathbf{where}\ f = e : [\overline{\forall \overline{a}_i.F_{TC_i}(\overline{u}^{i_n}) \sim \theta_i(u_{i_0})}^{m}, \forall \overline{a}.\theta(\overline{\pi}) \Rightarrow TC\ \overline{u}]}\ \textsc{Instance}$$

Figure 2. Declarative Type System

its own image, then the FD has no interpretation as a type-level function.

2. The specification does not accept *ambiguous* class or instance contexts ($unambig(\overline{b},\overline{a},\overline{\pi})$). Predicate *unambig* is defined as:

$$unambig(\overline{b},\overline{a},\overline{\pi}) \triangleq \overline{b} \subseteq dom(det(\overline{a},\overline{\pi}))$$

For class contexts this restriction ensures the well-formedness of the generated constraint schemes. Similarly, for instance contexts it ensures coherent semantics.

4.1.4 Determinacy Relation

The determinacy relation takes the form $det(\overline{a},\overline{\pi}) = \theta$ and can be read as *“Given known type variables $\overline{a}$ and a set of local class constraints $\overline{\pi}$, substitution θ maps type variables in $\overline{\pi}$ to equivalent types that draw type variables only from $\overline{a}$”.*

Formally, we define $det(\overline{a},\overline{\pi}) = \theta$ as $\overline{a};\overline{\pi} \vdash_{\mathsf{D}} \bullet \rightsquigarrow^{!} \theta$, where relation $\overline{a};\overline{\pi} \vdash_{\mathsf{D}} \theta_1 \rightsquigarrow \theta_2$ has a single rule:

$$\frac{\begin{array}{c}TC\ \overline{\tau} \in \overline{\pi} \qquad TC\ \overline{a} \mid a_{i_1} \ldots a_{i_n} \to a_{i_0} \\ fv(\tau_{i_0}) \nsubseteq \overline{a} \cup dom(\theta) \qquad fv(\tau_{i_1},\ldots,\tau_{i_n}) \subseteq \overline{a} \cup dom(\theta)\end{array}}{\overline{a};\overline{\pi} \vdash_{\mathsf{D}} \theta \rightsquigarrow [\overline{Proj_j^T(F_{TC_i}(\theta(\tau_{i_1}),\ldots,\theta(\tau_{i_n})))}/\overline{fv(\tau_{i_0})}] \cdot \theta}\ \textsc{Step}_D$$

We use the exclamation mark (!) to denote repeated applications of Rule $\textsc{Step}_D$, until it does not apply anymore. Note that if the superclass declarations of the program form a *Directed Acyclic Graph* (DAG), then this procedure is terminating.[7]

As an example of what the determinacy relation computes, consider the following example from Sulzmann et al. [32]:

$$\begin{array}{ll}\mathbf{class}\ G\ a\ b \mid a \to b & \mathbf{class}\ F\ a\ b \mid a \to b \\ \mathbf{class}\ H\ a\ b \mid a \to b & \mathbf{instance}\ (G\ a\ c, H\ c\ b) \Rightarrow F\ [a]\ [b]\end{array}$$

[7]Readers familiar with the work of Sulzmann et al. [32] will recognize that $dom(det(\overline{a},\overline{\pi})) = closure(\overline{a},\overline{\pi})$, where $closure(\cdot,\cdot)$ as defined in the *Refined Weak Coverage Condition* [32, Def. 15]. That is, it computes the set of determined variables of $\overline{\pi}$, along with a “proof” of their determinacy.

We compute the set of determined variables $det(a, \{G\ a\ c, H\ c\ b\})$ as follows:

$$\begin{array}{lll}\bullet & & \\ \rightsquigarrow & [F_G(a)/c] & (\text{from } (G\ a\ c)) \\ \rightsquigarrow & [F_H(F_G(a))/b, F_G(a)/c] & (\text{from } (H\ c\ b)) \\ \not\rightsquigarrow & & \end{array}$$

To illustrate what the projection type functions $Proj_i^T(\cdot)$ do, let us consider an alternative instance for F:

$$\mathbf{instance}\ (G\ a\ (c, Int), H\ c\ b) \Rightarrow F\ [a]\ [b]$$

In this case, we can no longer derive $c \sim F_G(a)$ but rather $(c, Int) \sim F_G(a)$. If we have a type-level function Fst available:

$$\mathbf{axiom}\ Fst\ a_1\ a_2 : Fst(a_1, a_2) \sim a_1$$

then c can be expressed in terms of a as: $c \sim Fst(F_G(a))$. In this case, $det(a, \{G\ a\ (c, Int), H\ c\ b\})$ proceeds as follows:

$$\begin{array}{lll}\bullet & \rightsquigarrow & [Fst(F_G(a))/c] & (\text{from } (G\ a\ (c, Int))) \\ & \rightsquigarrow & [F_H(Fst(F_G(a)))/b, Fst(F_G(a))/c] & (\text{from } (H\ c\ b)) \\ & \not\rightsquigarrow & & \end{array}$$

In general, a projection function $Proj_i^T(\cdot)$ is given by a single axiom

$$\mathbf{axiom}\ g\ \overline{a}^n : Proj_i^T(T\ a_1\ \ldots\ a_n) \sim a_i$$

As we illustrate in Appendix A.10, there is no need for such projection axioms, if we equip our system with *kind polymorphism* [36]. Yet, for simplicity we assume in the rest of the paper that such projection functions exist for all data types.

Notice that $det(\overline{\pi},\overline{a})$ can be slightly non-deterministic (multiple derivations for the same variable). For simplicity, we assume for the rest of the paper that it is deterministic, but return to this issue in Section 6.

5 Type Inference & Elaboration Into System F_C

This section exlains how to infer (principal) types for source language programs and how to elaborate them into System F_C at the same time.

$$
\begin{array}{lcll}
\upsilon & ::= & a \mid T \mid \upsilon_1\ \upsilon_2 \mid F(\overline{\upsilon}) \mid \forall a.\ \upsilon \mid \psi \Rightarrow \upsilon & \textit{type} \\[1ex]
\gamma & ::= & \langle \upsilon \rangle \mid \mathrm{sym}\ \gamma \mid \mathrm{left}\ \gamma \mid \mathrm{right}\ \gamma \mid \gamma_1 \mathbin{⨾} \gamma_2 \mid \psi \Rightarrow \gamma & \textit{coercion} \\
 & \mid & F(\overline{\gamma}) \mid \forall a.\ \gamma \mid \gamma_1[\gamma_2] \mid g\ \overline{\upsilon} \mid c \mid \gamma_1 @ \gamma_2 \mid \gamma_1\ \gamma_2 & \\[1ex]
\psi & ::= & \upsilon_1 \sim \upsilon_2 & \textit{proposition} \\[1ex]
t & ::= & x \mid K \mid \Lambda a.\ t \mid t\ \upsilon \mid \lambda(x : \upsilon).\ t \mid t_1\ t_2 \mid \Lambda(c : \psi).\ t & \textit{term} \\
 & \mid & t\ \gamma \mid t \triangleright \gamma \mid \mathbf{case}\ t_1\ \mathbf{of}\ p \to t_2 \mid \mathbf{let}\ x : \upsilon = t_1\ \mathbf{in}\ t_2 & \\[1ex]
p & ::= & K\ \overline{b}\ (\overline{c : \psi})\ (\overline{d : \tau})\ (\overline{f : \upsilon}) & \textit{pattern} \\[1ex]
\textit{decl} & ::= & \mathbf{data}\ T\ \overline{a}\ \mathbf{where}\ K : \upsilon & \textit{datatype declaration} \\
 & \mid & \mathbf{type}\ F(\overline{a}) & \textit{family declaration} \\
 & \mid & \mathbf{axiom}\ g\ \overline{a} : F(\overline{u}) \sim \upsilon & \textit{equality axiom} \\
 & \mid & \mathbf{let}\ x : \upsilon = t & \textit{value binding}
\end{array}
$$

Figure 3. System F$_C$ Syntax

5.1 Target Language: System F$_C$

Figure 3 presents the syntax of System F$_C$ [31], our target language. System F$_C$ extends the polymorphic lambda calculus [7] with first-class type equality proofs, called *coercions*.

Denoted by γ, coercions are evidence terms, encoding the proof tree for a type equality. Reflexivity $\langle \upsilon \rangle$, symmetry $(\mathrm{sym}\ \gamma)$ and transitivity $(\gamma_1 \mathbin{⨾} \gamma_2)$ express that type equality is an equivalence relation. Syntactic forms $F(\overline{\gamma})$ and $(\gamma_1\ \gamma_2)$ capture injection, while $(\mathrm{left}\ \gamma)$ and $(\mathrm{right}\ \gamma)$ capture projection, which follows from the injectivity of type application. Equality for universally quantified and qualified types is witnessed by forms $\forall a.\gamma$ and $\psi \Rightarrow \gamma$, respectively. Similarly, forms $\gamma_1[\gamma_2]$ and $\gamma_1 @ \gamma_2$ witness the equality of type instantiation or coercion application, respectively.

The most interesting forms are coercion variables c and coercion axioms $g\ \overline{\upsilon}$. The former represent *local constraints* [23, 34], which can be introduced via GADT [21] pattern-matching. The latter constitute the axiomatic part of the theory, and are generated from top-level axioms, which correspond to type family instances, newtype declarations [20], or, as we illustrate in Section 5.6.2, type class instances.

Since System F$_C$ is impredicative, the syntax of types υ does not discriminate between monotypes and type schemes. Yet, by convention, throughout the rest of the paper we use the metavariable τ to denote either source or System F$_C$ monotypes, since their syntax coincides. Similarly, we often use ϕ for propositions. Like in the source language, we elide all mention of kinds.

Expressions are standard, with the notable extensions of coercion abstraction $\Lambda(c : \psi).t$, coercion application $(t\ \gamma)$ and explicit type casting $(t \triangleright \gamma)$. In simple terms, if a term t has type υ_1 and γ is a witness of the equality $\upsilon_1 \sim \upsilon_2$, $(t \triangleright \gamma)$ has type υ_2. For the purpose of our work, it suffices to consider data types with a single data constructor, and case expressions with a single branch.

Declarations also appear in Figure 3. They include data type declarations, type family declarations, top-level equality axioms and value bindings.

We omit the type system of System F$_C$ from our main presentation. It can be found in [31] and is replicated in Appendix B.

5.2 Additional Constructs

During elaboration, we use the following additional constructs.

Type & Evidence Substitutions Much like HM(X) computes a type substitution when solving type constraints for refining as of yet unknown types, during inference we compute an *evidence substitution* η, for refining as of yet unknown class dictionaries d and type equality coercions c:

$$\eta ::= \bullet \mid [t/d] \cdot \eta \mid [\gamma/c] \cdot \eta$$

Type substitutions are standard, and map unification variables (denoted by Greek letters α and β) to monotypes:

$$\theta ::= \bullet \mid [\alpha/\tau] \cdot \theta$$

Evidence Annotations In order to perform type inference and elaboration into System F$_C$ simultaneously, we annotate all evidence types (equalities ϕ and class constraints π) with their corresponding System F$_C$ evidence variable, lifting the instance environment I to the program theory P:

$$P ::= \bullet \mid P, g\ \overline{a} : F(\overline{u}) \sim \tau \mid P, c : \phi \mid P, d : \pi \mid P, d : \forall \overline{a}.\overline{\pi} \Rightarrow TC\ \overline{u}$$

Simlarly for constraints:

$$
\begin{array}{lcll}
\mathcal{E} & ::= & \bullet \mid \mathcal{E}, c : \phi & \textit{annotated type equalities} \\
\mathcal{P} & ::= & \bullet \mid \mathcal{P}, d : \pi & \textit{annotated class constraints} \\
\mathcal{Q} & ::= & c : \phi \mid d : \pi & \textit{annotated type constraint} \\
\mathcal{C} & ::= & \bullet \mid \mathcal{C}, \mathcal{Q} & \textit{annotated type constraints}
\end{array}
$$

Match Contexts We also introduce *match contexts* $\mathbb{E}$, that is, nested case expressions with a hole.

$$\mathbb{E} ::= \square \mid \mathbf{case}\ d\ \mathbf{of}\ p \to \mathbb{E}$$

Match contexts are introduced via *dictionary destruction*, denoted as $\mathcal{P} \Downarrow \mathbb{E}$ which we define as follows:

$$\frac{}{\bullet \Downarrow \square}\ \textsc{Empty}$$

$$\frac{\begin{array}{c} K_{TC} : \forall \overline{a}\overline{b}.\overline{\psi} \Rightarrow \overline{\tau} \to \upsilon \to T\ \overline{a} \\ \overline{b'}, \overline{c}, \overline{d}, f\ \text{fresh} \qquad \theta = [\overline{\upsilon'}/\overline{a}, \overline{b'}/\overline{b}] \qquad \overline{d : \theta(\tau)}, \mathcal{P} \Downarrow \mathbb{E}_2 \\ \mathbb{E} = \mathbf{case}\ d_a\ \mathbf{of}\ K_{TC}\ \overline{b'}\ (\overline{c : \theta(\psi)})\ (\overline{d : \theta(\tau)})\ (f : \theta(\upsilon)) \to \mathbb{E}_2 \end{array}}{(d_a : T_{TC}\ \overline{\upsilon}'), \mathcal{P} \Downarrow \mathbb{E}}\ (\Downarrow)$$

Dictionary destruction $\mathcal{P} \Downarrow \mathbb{E}$ recursively pattern matches against class dictionaries $\mathcal{P}$ in a depth-first fashion, thus exposing all superclass constraints and FD-induced type equalities. In short, it computes the transitive closure of the superclass relation.

Throughout the rest of the paper we also denote the evidence or typing bindings introduced by a match context $\mathbb{E}$ as $P_{\mathbb{E}}$ or $\Gamma_{\mathbb{E}}$, respectively.

5.3 Term Elaboration

Figure 4 presents type inference and elaboration of terms into System F$_C$. The judgment takes the form $\Gamma \vdash_{\mathsf{tm}} e : \tau \rightsquigarrow t \mid \mathcal{P}; \mathcal{E}$. Given a typing environment Γ and a term e, it computes a set of *wanted* class constraints $\mathcal{P}$, a set of pending equality constraints $\mathcal{E}$, a monotype τ, and a System F$_C$ term t.

The most interesting rule is TmVar, which handles variables. We denote by $\overline{a}$ the type variables that appear in τ, and by $\overline{b}$ the ones that appear only in the context $\overline{\pi}$. The rule introduces *wanted* class constraints, appropriately instantiated with fresh unification and dictionary variables.

$\boxed{\Gamma \vdash_{\mathsf{tm}} e : \tau \leadsto t \mid \mathcal{P};\mathcal{E}}$ Term Elaboration

$$\frac{\begin{array}{c}(x : \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow \tau) \in \Gamma \\ \overline{\alpha}, \overline{d} \text{ fresh} \qquad \theta = [\overline{\alpha}/\overline{a}] \cdot det(\overline{\pi}, \overline{a})\end{array}}{\Gamma \vdash_{\mathsf{tm}} x : \theta(\tau) \leadsto x\ \overline{\alpha}\ (\theta(\overline{b}))\ \overline{d} \mid \overline{(d : \theta(\pi))}; \bullet}\ \textsc{TmVar}$$

$$\frac{\Gamma, x : \alpha \vdash_{\mathsf{tm}} e : \tau \leadsto t \mid \mathcal{P};\mathcal{E} \qquad \alpha \text{ fresh}}{\Gamma \vdash_{\mathsf{tm}} \lambda x.e : (\alpha \to \tau) \leadsto \lambda(x : \alpha).t \mid \mathcal{P};\mathcal{E}}\ \textsc{TmAbs}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{tm}} e_1 : \tau_1 \leadsto t_1 \mid \mathcal{P}_1;\mathcal{E}_1 \qquad \Gamma \vdash_{\mathsf{tm}} e_2 : \tau_2 \leadsto t_2 \mid \mathcal{P}_2;\mathcal{E}_2 \\ \alpha, c \text{ fresh} \qquad \mathcal{P} = \mathcal{P}_1, \mathcal{P}_2 \qquad \mathcal{E} = \mathcal{E}_1, \mathcal{E}_2, c : \tau_1 \sim \tau_2 \to \alpha\end{array}}{\Gamma \vdash_{\mathsf{tm}} e_1\ e_2 : a \leadsto (t_1 \triangleright c)\ t_2 \mid \mathcal{P};\mathcal{E}}\ \textsc{TmApp}$$

$$\frac{\begin{array}{c}\Gamma, x : \alpha \vdash_{\mathsf{tm}} e_1 : \tau_1 \leadsto t_1 \mid \mathcal{P}_1;\mathcal{E}_1 \\ \Gamma, x : \tau_1 \vdash_{\mathsf{tm}} e_2 : \tau_2 \leadsto t_2 \mid \mathcal{P}_2;\mathcal{E}_2 \\ \alpha, c \text{ fresh} \qquad \mathcal{P} = \mathcal{P}_1, \mathcal{P}_2 \qquad \mathcal{E} = \mathcal{E}_1, \mathcal{E}_2, c : \alpha \sim \tau_1\end{array}}{\Gamma \vdash_{\mathsf{tm}} (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \tau_2 \leadsto (\mathbf{let}\ x : \tau_1 = t_1\ \mathbf{in}\ t_2) \mid \mathcal{P};\mathcal{E}}\ \textsc{TmLet}$$

Figure 4. Term Elaboration

Notice that, unlike $\overline{a}$, $\overline{b}$ are not instantiated with fresh unification variables. Instead, we use the determinacy relation to express them in terms of $\overline{a}$. For example, given **class** $C\ a\ b \mid a \to b$, and $(x : C\ a\ b \Rightarrow a \to a) \in \Gamma$, we infer for x type $\alpha \to \alpha$, giving rise to the wanted constraint $C\ \alpha\ (F_C\ \alpha)$.

This treatment of $\overline{b}$ allows the unification algorithm of the next section to indirectly refine the non-parametric parameters of class constraints. Of course, this requires that x's signature is *unambiguous*, an issue we return to in Section 6.5.

Rule TmAbs is entirely standard.

Rule TmApp handles term applications $(e_1\ e_2)$. In addition to the constraints introduced by each subterm, we also require that $(\tau_1 \sim \tau_2 \to \alpha)$, like all HM(X)-based systems. In order to ensure that the elaborated term is well-typed, we explicitly cast e_1 with c, which serves as a placeholder for the equality proof computed by the *constraint entailment relation* (Section 5.5).

Rule TmLet handles (possibly recursive) let bindings. To simplify matters and as proposed by Vytiniotis et al. [33], we do not perform generalization.

5.4 Type Unification

We now turn to type unification in the presence of functional dependencies.

Type Reduction Judgment $P \vdash_{\mathsf{R}} \tau \leadsto \tau'; \gamma$ defines a (single-step) *type reduction relation* on monotypes, specified by the following rules.

$$\frac{P \vdash_{\mathsf{R}} \tau_1 \leadsto \tau_1'; \gamma}{P \vdash_{\mathsf{R}} \tau_1\ \tau_2 \leadsto \tau_1'\ \tau_2; \gamma\ \langle\tau_2\rangle}\ \textsc{Left}_R \qquad \frac{P \vdash_{\mathsf{R}} \tau_2 \leadsto \tau_2'; \gamma}{P \vdash_{\mathsf{R}} \tau_1\ \tau_2 \leadsto \tau_1\ \tau_2'; \langle\tau_1\rangle\ \gamma}\ \textsc{Right}_R$$

$$\frac{P \vdash_{\mathsf{R}} \tau_i \leadsto \tau_i'; \gamma_i \qquad \tau_j' = \tau_j, \forall j \neq i}{P \vdash_{\mathsf{R}} F(\overline{\tau}^n) \leadsto F(\overline{\tau}'^n); F(\langle\tau_1\rangle, \cdots \gamma_i, \cdots \langle\tau_n\rangle)}\ \textsc{Arg}_R$$

$$\frac{(g\ \overline{a} : F(\overline{u}) \sim \tau) \in P}{P \vdash_{\mathsf{R}} [\overline{\tau}/\overline{a}]F(\overline{u}) \leadsto [\overline{\tau}/\overline{a}]\tau; g\ \overline{\tau}}\ \textsc{Axiom}_R$$

We perform type reduction under program theory P, such that Rule Axiom$_R$ can expand type family applications when an appropriate axiom matches. We also annotate the reduction with a coercion γ, which witnesses the equality $\tau \sim \tau'$, as is required by the unification relation which we discuss next. It is straightforward to show that type reduction is sound:

Lemma 5.1 (Soundness of Type Reduction)**.** *If* $P \vdash_{\mathsf{R}} \tau_1 \leadsto \tau_2; \gamma$, *then* $P; fv(\tau_1) \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$.

Unification Type reduction is used by the *single-step unification relation* $P \vdash_{\mathsf{U}} c : \tau_1 \sim \tau_2 \leadsto \mathcal{E}; \theta; \eta$, which is given by rules

$$\frac{}{P \vdash_{\mathsf{U}} c : \tau \sim \tau \leadsto \bullet; \bullet; [\langle\tau\rangle/c]}\ \textsc{Refl}_U$$

$$\frac{P \vdash_{\mathsf{U}} c' : \tau_2 \sim \tau_1 \leadsto \mathcal{E}; \theta; \eta \qquad c' \text{ fresh}}{P \vdash_{\mathsf{U}} c : \tau_1 \sim \tau_2 \leadsto \mathcal{E}; \theta; \eta \cdot [\mathrm{sym}\ c'/c]}\ \textsc{Sym}_U$$

$$\frac{\alpha \notin fv(\tau)}{P \vdash_{\mathsf{U}} c : \alpha \sim \tau \leadsto \bullet; [\tau/\alpha]; [\langle\tau\rangle/c]}\ \textsc{Var}_U$$

$$\frac{P \vdash_{\mathsf{R}} F(\overline{\tau}) \leadsto \tau_2; \gamma \qquad c' \text{ fresh}}{P \vdash_{\mathsf{U}} c : F(\overline{\tau}) \sim \tau_1 \leadsto \{c' : \tau_2 \sim \tau_1\}; \bullet; [\gamma \fatsemi c'/c]}\ \textsc{Red}_U$$

$$\frac{c_1, c_2 \text{ fresh} \qquad \gamma = c_1\ c_2}{P \vdash_{\mathsf{U}} c : \tau_1\ \tau_2 \sim \tau_1'\ \tau_2' \leadsto \{c_1 : \tau_1 \sim \tau_1', c_2 : \tau_2 \sim \tau_2'\}; \bullet; [\gamma/c]}\ \textsc{App}_U$$

In layman's terms, the judgment holds for an equality $\tau_1 \sim \tau_2$ iff the unification problem can be reduced to a simpler unification problem for the set of equality constraints $\mathcal{E}$ and type substitution θ. Since in our target language casting needs explicit equality proofs, we also accumulate an evidence substitution η, which explains how evidence from solving $\mathcal{E}$ can be used to construct evidence c for $\tau_1 \sim \tau_2$.

5.5 Constraint Entailment

Single-step constraint entailment takes the form $P \vdash_{\mathsf{E}} Q \leadsto C; \theta; \eta$ and simplifies a constraint Q to a set of simpler C and a type substitution θ. Additionally, it computes an evidence substitution η, which maps evidence variables (coercion or dictionary variables) to evidence terms composed by the simpler evidence. The relation is given by the following rules:

$$\frac{(d' : \forall \overline{a}.\overline{\pi} \Rightarrow TC\ \overline{u}) \in P \qquad \overline{d} \text{ fresh} \qquad t = d'\ \overline{\tau}\ \overline{d}}{P \vdash_{\mathsf{E}} d : [\overline{\tau}/\overline{a}](TC\ \overline{u}) \leadsto (\overline{d : [\overline{\tau}/\overline{a}]\pi}); \bullet; [t/d]}\ \textsc{Cls}_E$$

$$\frac{P \vdash_{\mathsf{U}} c : \tau_1 \sim \tau_2 \leadsto \mathcal{E}; \theta; \eta}{P \vdash_{\mathsf{E}} c : \tau_1 \sim \tau_2 \leadsto \mathcal{E}; \theta; \eta}\ \textsc{Eq}_E$$

$$\frac{P \vdash_{\mathsf{R}} \tau_i \leadsto \tau_i'; \gamma_i \qquad \forall i \in [1 \ldots n] \qquad d' \text{ fresh}}{P \vdash_{\mathsf{E}} (d : TC\ \overline{\tau}^n) \leadsto (d' : TC\ \overline{\tau}'^n); \bullet; [d' \triangleright T_{TC}\ \overline{\mathrm{sym}\ \gamma_i}^n/d]}\ \textsc{Red}_E$$

Our system needs to handle both class and equality constraints, which is reflected in the rules: Rule Cls$_E$ formalizes the standard SLD resolution (backwards chaining), Rule Eq$_E$ performs single-step unification on equality constraints, and Rule Red$_E$ allows for type reduction on class parameters.

By repeatedly applying single-step constraint entailment, we obtain the reflexive and transitive closure $P \vdash_{\mathsf{E}} C \leadsto^* C; \theta; \eta$ (see Appendix A for its formal definition). We denote the case when C cannot be further reduced as $P \vdash_{\mathsf{E}} C \leadsto^! C'; \theta; \eta$. To ensure that type inference is decidable, it is essential that constraint entailment is terminating; Section 6.1 provides sufficient conditions.

$\boxed{\Gamma \vdash_{\mathtt{cls}} cls \rightsquigarrow \overline{decl} \mid \Gamma_c}$ Class Elaboration

$$\frac{\begin{array}{c} unambig(\overline{b}, \overline{a}, \overline{\pi}) \qquad \Gamma, \overline{a} \vdash_{\mathtt{ty}} \sigma \rightsquigarrow \upsilon \qquad \overline{\Gamma, \overline{a}, \overline{b} \vdash_{\mathtt{cc}} \pi \rightsquigarrow \tau} \qquad \psi_i = F_{TC_i}(\overline{a}^{i_n}) \sim a_{i_0} \qquad fd_i \equiv \overline{a}^{i_n} \to a_{i_0} \\ \overline{decl_c} = [\mathbf{data}\ T_{TC}\ \overline{a}\ \mathbf{where}\ \{K_{TC} : \forall \overline{a}\overline{b}.\overline{\psi} \Rightarrow \overline{\tau} \to \upsilon \to T_{TC}\ \overline{a}\ \}, \overline{\mathbf{type}\ F_{TC_i}\ \overline{a}^{i_n}}^{m}, f = \Lambda\overline{a}.\lambda(d : T_{TC}\ \overline{a}).\mathbf{case}\ d\ \mathbf{of}\ \{K_{TC}\ \overline{b}\ \overline{c}\ \overline{d}\ x \to x\}] \end{array}}{\Gamma \vdash_{\mathtt{cls}} \mathbf{class}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{a} \mid \overline{fd}^{m}\ \mathbf{where}\ f :: \sigma \rightsquigarrow \overline{decl_c} \mid [f : \forall \overline{a}.TC\ \overline{a} \Rightarrow \sigma]}\ \textsc{Class}$$

$\boxed{P; \Gamma \vdash_{\mathtt{inst}} inst \rightsquigarrow \overline{decl} \mid P_i}$ Instance Elaboration

$$\frac{\begin{array}{c} unambig(\overline{b}, \overline{a}, \overline{\pi}) \qquad (\overline{d : \pi}) \Downarrow \mathbb{E} \qquad d_I, \overline{d}\ \text{fresh} \qquad P_I = P, P_{ax}, \overline{d : \pi}, P_{\mathbb{E}} \qquad \Gamma_I = \Gamma, \overline{a}, \overline{b}, \Gamma_{\mathbb{E}} \qquad S_I = \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u} \\ \overline{\Gamma, \overline{a}, \overline{b} \vdash_{\mathtt{cc}} \pi \rightsquigarrow \tau} \qquad (f : \forall \overline{a}'.TC\ \overline{a}' \Rightarrow \sigma) \in \Gamma \qquad P_I, d_I : S_I; \Gamma_I \vdash_{\mathtt{tm}} e : [\overline{u}/\overline{a'}]\sigma \rightsquigarrow t \qquad S_I \hookrightarrow P_{ax} \qquad P_I \vdash_{\mathtt{sc}} TC\ \overline{u} \rightsquigarrow (\overline{\tau}_b, \overline{t}_d, \overline{\gamma}_c) \end{array}}{P; \Gamma \vdash_{\mathtt{inst}} \mathbf{instance}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u}\ \mathbf{where}\ f = e \rightsquigarrow [\overline{\mathbf{axiom}\ P_{ax}}, d_I = \Lambda\overline{a}\overline{b}.\lambda\overline{(d : \tau)}.\mathbb{E}[K_{TC}\ \overline{\tau}_b\ \overline{t}_d\ \overline{\gamma}_c\ t]] \mid [P_{ax}, d_I : \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u}]}\ \textsc{Instance}$$

Figure 5. Declaration Elaboration

5.6 Declaration Elaboration

Finally, Figure 5 presents elaboration of declarations. Only elaboration of class and instance declarations is given, since the other cases are entirely standard (see Appendix A).

5.6.1 Elaboration of Class Declarations

Class elaboration takes the form $\Gamma \vdash_{\mathtt{cls}} cls \rightsquigarrow \overline{decl} \mid \Gamma_c$ and is given by a single rule, Rule Class.

The encoding of a class constraint in System F$_C$ is that of a GADT-dictionary [21], such that we can store existentially quantified variables $\overline{b}$, as well as the local constraints ψ_i, each corresponding to a functional dependency annotation.

In contrast to earlier formalizations of type classes, checking a class declaration does not give rise to a direct extension of the program theory. While Equation CS1a may have a direct interpretation as a System F$_C$ term, Equation CS1b does not: System F$_C$ does not support functions that return coercions; this would not be compatible with System F$_C$'s coercion erasure and a call-by-need semantics.

Instead, both schemes can be uniformly elaborated as *match contexts*. For example, the following match context corresponds to the logical implication $Ord\ a \Rightarrow Eq\ a$:

$$\mathbb{E} = \mathbf{case}\ d_{Ord}\ \mathbf{of}\ \{\ K_{Ord}\ d_{Eq}\ f \to \square\ \}$$

We reject unconditionally ambiguous class declarations, via restriction $unambig(\overline{b}, \overline{a}, \overline{\pi})$.

5.6.2 Elaboration of Class Instances

Instance elaboration also appears in Figure 5 and takes the form $P; \Gamma \vdash_{\mathtt{inst}} inst \rightsquigarrow \overline{decl} \mid P_i$. That is, an instance declaration $inst$ is elaborated to System F$_C$ declarations $\overline{decl}$ and gives rise to the program theory extension P_i. To aid readability, we formalize instance elaboration by means of the following auxiliary relations:

Axiom Generation As we explained in Section 3, each class instance gives rise to a type family axiom for every functional dependency of the class. This semantics is reflected in Equation CS2b, and directly corresponds to axioms P_{ax}, as produced by relation:

$$\frac{\begin{array}{c} (fd_i \equiv \overline{a}^{i_n} \to a_{i_0}) \in (\overline{fd}^{m} \in TC) \\ \theta_i = det(fv(\overline{u}^{i_n}), \overline{\pi}) \qquad fv(\theta_i(u_{i_0})) \subseteq fv(\overline{u}^{i_n}) \qquad \overline{g}\ \text{fresh} \end{array}}{(\forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u}) \hookrightarrow \overline{g_i\ (fv(\overline{u}^{i_n})) : F_{TC_i}(\overline{u}^{i_n}) \sim \theta_i(u_{i_0})}^{m}}\ \textsc{AxGen}$$

Premise $fv(\theta_i(u_{i_0})) \subseteq fv(\overline{u}^{i_n})$ ensures that the generated axioms are well-formed; like the Liberal Coverage Condition [32] it checks that the image of every functional dependency is determined by its domain.

Method Translation & Type Subsumption Since method implementations are in effect explicitly typed, we need a procedure for deciding *type subsumption*. We say that a polytype σ_1 *subsumes* polytype σ_2, if any expression that can be assigned type σ_1 can also be assigned type σ_2. Since we elaborate during inference, we perform type inference and the subsumption check simultaneously, by means of relation $P; \Gamma \vdash_{\mathtt{tm}} e : \sigma \rightsquigarrow t$, which is given by rule:

$$\frac{\begin{array}{c} \Gamma \vdash_{\mathtt{tm}} e : \tau_1 \rightsquigarrow t \mid \mathcal{P}; \mathcal{E} \\ \Gamma \vdash_{\mathtt{ty}} (\forall \overline{a}.\overline{\pi} \Rightarrow \tau_2) \qquad \overline{\Gamma \vdash_{\mathtt{cc}} \pi \rightsquigarrow \tau} \qquad c, \overline{d}\ \text{fresh} \\ (\overline{d : \pi}) \Downarrow \mathbb{E} \qquad P, (\overline{d : \pi}), P_{\mathbb{E}} \vdash_{\mathtt{E}} \mathcal{P}, \mathcal{E}, (c : \tau_1 \sim \tau_2) \rightsquigarrow^{!} \bullet; \theta; \eta \end{array}}{P; \Gamma \vdash_{\mathtt{tm}} e : (\forall \overline{a}.\overline{\pi} \Rightarrow \tau_2) \rightsquigarrow \Lambda\overline{a}.\lambda(\overline{d : \tau}).\mathbb{E}[\eta(\theta(t \triangleright c))]}\ (\leq)$$

In short, from the assumption $\overline{\pi}$ we need to be able to completely derive all constraints that arise from typing e and the equality $(\tau_1 \sim \tau_2)$. We locally extend the program theory with the transitive closure of the superclass relation on $\overline{\pi}$, thus exposing both superclass dictionaries and FD constraints induced by $\overline{\pi}$.

Superclass Entailment Furthermore, we need to ensure that the instance context $\overline{\pi}$ (along with the newly created axioms P_{ax}) completely entails the superclass and FD constraints. This procedure is captured by relation $P_{inst} \vdash_{\mathtt{sc}} (TC\ \overline{u}) \rightsquigarrow (\overline{\tau}, \overline{t}, \overline{\gamma})$:

$$\frac{\begin{array}{c} \mathbf{class}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{a} \mid \overline{fd}^{m} \\ \overline{c}, \overline{d}\ \text{fresh} \qquad \theta = [\overline{u}/\overline{a}] \cdot det(\overline{\pi}, \overline{a}) \\ P_{inst} \vdash_{\mathtt{E}} (\overline{d : \theta(\pi)}), (\overline{c : \theta(F_{TC_i}(\overline{a}^{i_n}) \sim a_{i_0})}) \rightsquigarrow^{!} \bullet; \theta_s; \eta_s \end{array}}{P_{inst} \vdash_{\mathtt{sc}} (TC\ \overline{u}) \rightsquigarrow (\theta_s(\theta(\overline{b})), \eta_s(\overline{d}), \eta_s(\overline{c}))}\ \text{SC}$$

Notice that the relation also computes the existential types introduced in the superclass context $\theta_s(\theta(\overline{b}))$, which should also be stored in the resulting GADT dictionary.

Instance Elaboration Finally, Rule Instance utilizes the above relations to produce the dictionary transformer d_I, which reflects the *Instance Constraint Scheme* (Equation CS2a).

Since we do not encode the superclass relation using constraint schemes but via match contexts, both the method elaboration and the superclass entailment are performed under environment P_{all}, which includes not only the instance context $\overline{\pi}$ and axioms P_{ax}, but also the transitive closure of the superclass relation P', obtained by exhaustively destructing assumptions $\overline{\pi}$.

6 Metatheory

This section considers the key meta-theoretical properties of both the type system and the type inference & elaboration algorithm.

6.1 Termination of Type Inference

Our *Termination Conditions* ensure termination of type inference:

(a) The superclass relation forms a *directed acyclic graph* (DAG).

(b) In each class instance (**instance** $\forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u}$):
- no variable has more occurrences in a type class constraint π than the head $(TC\ \overline{u})$, and
- each class constraint π in the context $\overline{\pi}$ has fewer constructors and variables (taken together, counting repetitions) than the head $(TC\ \overline{u})$.

(c) For every generated axiom $(g\ \overline{a} : F(\overline{u}) \sim \tau)$, in every subterm $(F_1(\overline{\tau}_1) \subseteq \tau)$:
- there is no subterm $(F_2(\overline{\tau}_2) \subseteq F_1(\overline{\tau}_1))$,
- the sum of the number of type constructors and type variables is smaller than the corresponding number in $\overline{u}$, and
- there are not more occurrences of any variable a than in $\overline{u}$.

The first restriction ensures that relations $det(\overline{\pi}, \overline{a})$ and $(\mathcal{P} \Downarrow \mathbb{E})$ terminate, since they both compute the transitive closure of the superclass relation.

The second restriction, borrowed from the Paterson Conditions [32, Def. 11], ensures that instance contexts are *decreasing*, so that class resolution (Rules Cls_E and Rules Red_E) is also terminating, given that the type equality axioms are strongly normalizing.

Lastly, the third restriction (borrowed from Schrijvers et al. [22, Def. 5]) ensures that the generated axioms are strongly normalizing, that is, confluent and terminating, which allows us to turn constraint entailment (Section 5.5) into a deterministic function.

Theorem 6.1. *If a program satisfies the Termination Conditions, then type inference terminates.*

6.2 Functional Dependency Property

There are two important properties that regulate the functional dependency property.

Compatibility Firstly, we need to make sure that there are no two conflicting definitions that associate two different values with the same key. To this end, we impose the *Compatibility Condition*:

Definition 6.2 (Compatibility Condition)**.** Let there be a class declaration and any pair of instance declarations for that class:

$$\begin{array}{l}\textbf{class}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{a} \mid fd_1, \ldots, fd_m\ \textbf{where}\ f :: \sigma \\ \textbf{instance}\ \forall \overline{a}_1\overline{b}_1.\overline{\pi}_1 \Rightarrow TC\ \overline{u}_1\ \textbf{where}\ f = e_1 \\ \textbf{instance}\ \forall \overline{a}_2\overline{b}_2.\overline{\pi}_2 \Rightarrow TC\ \overline{u}_2\ \textbf{where}\ f = e_2\end{array}$$

Then, for each functional dependency $fd_i \equiv a_{i_1}, \ldots, a_{i_n} \rightarrow a_{i_0}$ the following should hold:

$$compat(F_{TC_i}(\overline{u}_1^{i_n}) \sim \theta_{i1}(u_{i_0 1}), F_{TC_i}(\overline{u}_2^{i_n}) \sim \theta_{i2}(u_{i_0 2}))$$

where $\theta_{i1} = det(fv(\overline{u}_1^{i_n}), \overline{\pi}_1)$ and $\theta_{i2} = det(fv(\overline{u}_2^{i_n}), \overline{\pi}_2)$.

Relation $compat(\cdot, \cdot)$ is the *compatibility relation*, as defined by Eisenberg et al. [5]:

Definition 6.3 (Compatibility)**.** Two equalities $\phi_1 = F(\overline{u}_1) \sim \tau_1$ and $\phi_2 = F(\overline{u}_1) \sim \tau_2$ are compatible –denoted as $compat(\phi_1, \phi_2)$– iff $unify(\overline{u}_1, \overline{u}_2) = \theta$ implies $\theta(\tau_1) = \theta(\tau_2)$.

In the nomenclature of Jones [15, Section 6.1] and Sulzmann et al. [32, Def. 6–8] compatibility is known as *consistency*. Both works impose a very conservative consistency condition, which requires, for any two instance heads $(TC\ \overline{u}_1)$ and $(TC\ \overline{u}_2)$ and any functional dependency $fd_i \equiv \overline{a}^{i_n} \rightarrow a_{i_0}$ of class TC, that $\theta(u_{i_0 1}) = \theta(u_{i_0 2})$ if $unify(\overline{u}_1^{i_n}, \overline{u}_2^{i_n}) = \theta$. This means that the function is fully determined by the instance head, and cannot depend on the instance context. The latter is supported by our more liberal Compatibility Condition, which meets Section 2.2's Challenge 1, by providing a criterion to verify the consistency of more liberal instances.

Notice that both Jones and Sulzmann et al. consider an additional property, *coverage*, which stipulates that the image of every functional dependency instance is fully determined by its domain. Jones enforces this property through a conservative condition, while Sulzmann et al. consider the more liberal *Liberal Coverage Condition* [32, Def. 15] which also takes the instance context into account. Our system does not require an external coverage condition as it already internalizes coverage in the determinacy relation (Section 4.1.4).

Unambiguous Witness Functions Secondly, we need to make sure that the function that witnesses the functional dependency is uniquely determined. For this reason, we impose the Unambiguous Witness Condition.

Definition 6.4 (Unambiguous Witness)**.** Let there be a class declaration and any instance for that class:

$$\begin{array}{l}\textbf{class}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{a} \mid \overline{fd}^{m}\ \textbf{where}\ f :: \sigma \\ \textbf{instance}\ \forall \overline{a}'\overline{b}'.\overline{\pi}' \Rightarrow TC\ \overline{u}\ \textbf{where}\ f = e\end{array}$$

Then, for each functional dependency $fd_i \equiv \overline{a}^{i_n} \rightarrow a_{i_0}$ it is required that $det(\overline{\pi}', fv(\overline{u}^{i_n}))$ is non-ambiguous on $fv(u_{i_0})$.

A witness derivation $det(\overline{\pi}, \overline{a}) = \theta$ is non-ambiguous *on type variables* $\overline{b}$ iff $\overline{b} \subseteq dom(\theta)$ and $\theta(\overline{b})$ is independent of the order in which relation $\overline{a}; \overline{\pi} \models \theta_1 \rightsquigarrow \theta_2$ selects class constraints π from $\overline{\pi}$.

To see why this condition is important, consider for example the following declarations:

$$\begin{array}{ll}\textbf{class}\ C_1\ a\ b \mid a \rightarrow b & \textbf{class}\ C\ a\ b \mid a \rightarrow b \\ \textbf{class}\ C_2\ a\ b \mid a \rightarrow b & \textbf{instance}\ (C_1\ a\ b, C_2\ a\ b) \Rightarrow C\ [a]\ [b]\end{array}$$

What axiom should the C instance give rise to? Using the instance context $(C_1\ a\ b, C_2\ a\ b)$, we can derive either of the two:

$$\begin{array}{l}\textbf{axiom}\ g_1\ a : F_C\ a \sim [F_{C_1}\ a] \\ \textbf{axiom}\ g_2\ a : F_C\ a \sim [F_{C_2}\ a]\end{array}$$

Yet, depending on the choice, different programs are accepted. For example, from the given constraints $\{C_1\ a\ b, C_2\ a\ c, C\ [a]\ [b]\}$ we can derive $(b \sim c)$ if g_1 is available; the same does not hold for g_2.

Even worse, the choice of the axiom affects the compatibility of the instance with other instances for the same class. To support modular compilation, we cannot optimise the choice by taking the rest of the program into account.

For these reasons, our Unambiguous Witness Condition rejects programs with such ambiguity. Another solution would be for the programmer to manually resolve the ambiguity by expressing a preference.

6.3 Type Substitution Property

Our system satisfies the *type substitution property*.

Theorem 6.5. *If $I; \Gamma \vdash_{\mathsf{tm}} e : \forall a.\sigma$ and $\Gamma \vdash_{\mathsf{ty}} \tau$, then $I; \Gamma \vdash_{\mathsf{tm}} e : [\tau/a]\sigma$.*

In short, a type system satisfies the type substitution property iff typing a term e with type $\forall a.\sigma$ implies that we can also type it with the instantiated type $[\tau/a]\sigma$.

Chakravarty et al. [1] used the following example to compare three systems (the system implemented by the Hugs compiler, the system implemented by GHC, and a system designed by Stuckey and Sulzmann [30]) with respect to whether they satisfy this property.

$$\begin{array}{l} \textbf{class}\ C\ a\ b \mid a \to b\ \textbf{where}\ \{\ \mathit{foo} :: a \to b\ \} \\ \textbf{instance}\ C\ \mathit{Bool}\ \mathit{Int}\ \textbf{where}\ \{\ \mathit{foo} = \ldots\ \} \end{array}$$

Three possible signatures for function *bar* are:

$$\begin{array}{llll} \mathit{bar} :: C\ a & b \Rightarrow a & \to b & \text{(1: most general type)} \\ \mathit{bar} :: C\ \mathit{Bool}\ b & \Rightarrow \mathit{Bool} & \to b & \text{(2: substitution instance)} \\ \mathit{bar} :: & \mathit{Bool} & \to \mathit{Int} & \text{(3: apply the fd)} \\ \mathit{bar} = \mathit{foo} & & & \end{array}$$

All three signatures are accepted by the system of Stuckey and Sulzmann, which is based on Constraint Handling Rules. Yet, signature (2) is rejected by both GHC and Hugs, which use a dictionary-based translation to an explicitly-typed language based on System F. Our system accepts all three signatures.

As far as we know, our system is the first with functional dependencies to satisfy the type substitution property, while translating to a typed intermediate language.

6.4 Algorithm Soundness

The algorithm performs two tasks at once: type inference and elaboration into System F_C. It is sound on both accounts.

Firstly, the type inference task is sound.

Theorem 6.6 (Soundness of Type Inference)**.** *If $\vdash_{\mathsf{pgm}} pgm \rightsquigarrow \overline{decl}$, then $\vdash_{\mathsf{pgm}} pgm$.*

Secondly, the elaboration produces well-typed System F_C code.

Theorem 6.7 (Preservation of Typeability Under Elaboration)**.** *If $\vdash_{\mathsf{pgm}} pgm \rightsquigarrow \overline{decl}$, then $\vdash^{\mathsf{fc}}_{\mathsf{pgm}} \overline{decl}$.*

Moreover, to be type safe, System F_C requires the consistency of the axiomatic equational theory. This property follows from the Compatibility Condition:

Theorem 6.8 (Consistency of Elaborated Programs)**.** *If pgm satisfies the Compatibility Condition and $\vdash_{\mathsf{pgm}} pgm \rightsquigarrow \overline{decl}$, then the top-level typing environment of $\overline{decl}$ is consistent (according to the definition of System F_C consistency [31]).*

6.5 Ambiguity

Following the Haskell tradition, we also require that *unconditionally ambiguous* type signatures are rejected, since the runtime behavior of terms that inhabit them is not well-specified. Checking signatures for ambiguity is straightforward:

Theorem 6.9 (Non-ambiguous Types)**.** *Let there be a (well-scoped) type $\sigma = \forall \overline{a}.\overline{\pi} \Rightarrow \tau$. Iff $fv(\overline{\pi}) \subseteq dom(det(fixed(\tau), \overline{\pi})) \cup fixed(\tau)$, then σ is unambiguous.*

Function $\mathit{fixed}(\cdot)$ computes the set of *fixed* variables of a monotype:

$$\begin{array}{lclclcl} \mathit{fixed}(a) & = & \{a\} & \quad & \mathit{fixed}(\tau_1\ \tau_2) & = & \mathit{fixed}(\tau_1) \cup \mathit{fixed}(\tau_2) \\ \mathit{fixed}(T) & = & \emptyset & & \mathit{fixed}(F(\overline{\tau})) & = & \emptyset \end{array}$$

Intuitively, all type variables appearing in the context $\overline{\pi}$ should be determined from the monotype τ, either by directly appearing in τ, or indirectly via a functional dependency (or a chain of them). For instance, given the class declaration **class** $C\ a\ b \mid a \to b$ we conclude that signature $C\ a\ b \Rightarrow a \to a$ is unambiguous because type variable b is functionally determined by a.

6.6 Principality of Types

The specification of Section 4 has the *principal type property*:

Theorem 6.10 (Principal Types)**.** *If e is well-typed, then there exists a type σ_0 (the principal type), such that $I; \Gamma \vdash_{\mathsf{tm}} e : \sigma_0$ and, for all σ such that $I; \Gamma \vdash_{\mathsf{tm}} e : \sigma$, we have that $I \models \sigma_0 \preceq \sigma$.*

Here relation $I \models \sigma_0 \preceq \sigma$ defines *type subsumption*:

$$\frac{I, [\overline{T}/\overline{b}]\overline{\pi}_2 \models \overline{\pi}_1, \tau_1 \sim [\overline{T}/\overline{b}]\tau_2 \qquad \overline{T}\ \text{fresh type constructors}}{I \models (\forall \overline{a}.\overline{\pi}_1 \Rightarrow \tau_1) \preceq (\forall \overline{b}.\overline{\pi}_2 \Rightarrow \tau_2)}\ (\preceq)$$

Moreover, without introducing further formal notation, we state that type inference derives the principal type:

Theorem 6.11 (Inference Computes Principal Types)**.** *The type inference of Section 5 computes only principal types.*

6.7 Coherence

Another crucial property of our system is *coherence*: every different valid type derivation for a program should lead to a resulting program that has the same dynamic semantics. To ensure this, it is sufficient to restrict ourselves to *non-overlapping instances*:

Definition 6.12 (Non-overlapping Instances)**.** Any two instance heads $(TC\ \overline{u}_1)$ and $(TC\ \overline{u}_2)$ for the same class should not overlap $(\nexists\theta.\theta(\overline{u}_1) = \theta(\overline{u}_2))$.

6.8 Completeness

Finally, we conjecture that our algorithm is complete with respect to the declarative type system for programs that satisfy the Termination and Unambiguous Witness Conditions.

Conjecture 6.13 (Completeness of Type Inference)**.** *If $\vdash_{\mathsf{pgm}} pgm$, then $\vdash_{\mathsf{pgm}} pgm \rightsquigarrow \overline{decl}$.*

7 Related Work

Functional Dependencies Functional dependencies were introduced in Haskell's class system by Jones [15], and its first sound and decidable type inference has been given by Duck et al. [4]. Follow-up work by Sulzmann et al. [32] formalized functional dependencies in terms of *Constraint Handling Rules* [6], and thoroughly studied several extensions, including multi-range functional dependencies and weakened variants of the coverage condition.

Non-Functional Improvement Our work treats functional dependencies as type-level functions, as originally intended by Jones [15]. Alternatively, one can view functional dependencies as a more general mechanism for guiding type inference, e.g., as asked for in GHC feature request #8634. Under this interpretation, the domain of a functional dependency does not necessarily determine its image; the result can also be partially determined. This fits with Jones' more general theory of *improvement* [14]. A flexible system for improvement was presented by Stuckey and Sulzmann [30], where the programmer can extend type inference, including partial improvements, directly through Constraint Handling Rules. It is an open question how to integrate this approach with a typed intermediate language.

Type Families Functional Dependencies are most closely related to associated type synonyms [1, 22, 25] and our formalisation is based on theirs. This enables a more direct comparison and integration of both features in future work.

Our Compatibility Condition is based on that of Eisenberg et al. [5] for open type families, which relaxes the non-overlapping check of Schrijvers et al. [22].

Injective Associated Type Synonyms Stolarek et al. [29] proposed injectivity annotations for type families, which closely resemble functional dependencies in both syntax and semantics: to indicate that a type family with result type b is injective in the i-th argument a_i a user can add an annotation $b \to a_i$. They introduce a new System F_C coercion form to witness injectivity. Our elaboration does not need this new form, as the type class dictionary serves to hold the witness; this approach could be used for the injectivity of associated type synonyms too.

Expanding on an earlier sketch by Schrijvers and Sulzmann [24], Serrano et al. [26] discuss an approach for elaborating type classes into type families augmented with dictionaries. They use injectivity annotations to elaborate functional dependencies.

8 Conclusion

This paper has tackled a number of important open challenges concerning functional dependencies: We have provided a declarative type system that explicitly reconstructs the implicit function that witnesses the functional dependency. Alongside with the declarative type system we have presented a type inference algorithm that uses the same building blocks as that of associated type synonyms and the first elaboration of functional dependencies into a typed intermediate language, System F_C.

We believe that our work enables the proper integration of functional dependencies in Haskell's ecosystem of advanced type-level features. In future work feature requests like #11534 can be addressed, and the reconstructed witness function can be exposed to the user for explicit use.

References

[1] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated Type Synonyms. *SIGPLAN Not.* 40, 9 (Sept. 2005), 241–253.

[2] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. 2005. Associated Types with Class. *SIGPLAN Not.* 40, 1 (Jan. 2005), 1–13.

[3] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for Fine-grained Reuse. *ACM Trans. Program. Lang. Syst.* 28, 2 (March 2006), 331–388.

[4] Gregory J. Duck, Simon Peyton-Jones, Peter J. Stuckey, and Martin Sulzmann. 2004. Sound and Decidable Type Inference for Functional Dependencies. In *Programming Languages and Systems*, David Schmidt (Ed.). Lecture Notes in Computer Science, Vol. 2986. Springer Berlin Heidelberg, 49–63.

[5] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. *SIGPLAN Not.* 49, 1 (Jan. 2014), 671–683.

[6] Thom W. Frühwirth. 1995. Constraint Handling Rules. In *Selected Papers from Constraint Programming: Basics and Trends*. Springer-Verlag, London, UK, 90–107.

[7] Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press, New York, NY, USA.

[8] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: Linguistic Support for Generic Programming in C++. *SIGPLAN Not.* 41, 10 (Oct. 2006), 291–310.

[9] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996), 109–138.

[10] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. 1996. *The Mercury Language Reference Manual*. Technical Report.

[11] Mark Jones. 2010. *The Habit Programming Language: The Revised Preliminary Report*.

[12] Mark P. Jones. 1992. A theory of qualified types. In *ESOP '92*, Bernd Krieg-Brückner (Ed.). Lecture Notes in Computer Science, Vol. 582. Springer Berlin Heidelberg, 287–306.

[13] Mark P. Jones. 1993. A System of Constructor Classes: Overloading and Implicit Higher-order Polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 52–61.

[14] Mark P. Jones. 1995. Simplifying and Improving Qualified Types. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, New York, NY, USA, 160–169.

[15] Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Programming Languages and Systems*, Gert Smolka (Ed.). Lecture Notes in Computer Science, Vol. 1782. Springer Berlin Heidelberg, 230–244.

[16] Mark P. Jones and Iavor S. Diatchki. 2008. Language and Program Design for Functional Dependencies. *SIGPLAN Not.* 44, 2 (Sept. 2008), 87–98.

[17] Simon Peyton Jones. 1997. Bulk Types With Class. In *Proceedings of the Second Haskell Workshop*.

[18] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. ACM, New York, NY, USA, 96–107.

[19] The Coq development team. 2004. *The Coq proof assistant reference manual*. LogiCal Project. http://coq.inria.fr Version 8.0.

[20] Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

[21] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. *SIGPLAN Not.* 41, 9 (Sept. 2006), 50–61.

[22] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type checking with open type functions. *ACM Sigplan Notices* 43, 9 (Sep 2008), 51–62.

[23] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and Decidable Type Inference for GADTs. *SIGPLAN Not.* 44, 9 (Aug. 2009), 341–352.

[24] Tom Schrijvers and Martin Sulzmann. 2008. Unified type checking for classes and type families. (2008). https://lirias.kuleuven.be/handle/123456789/186697

[25] Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel Chakravarty. 2007. Towards open type functions for Haskell. In *Preproceedings of Implementation and Application of Functional Languages*, O. Chitil (Ed.). Computing Laboratory, University of Kent, 233–251.

[26] Alejandro Serrano, Jurriaan Hage, and Patrick Bahr. 2015. Type Families with Class, Type Classes with Family. *SIGPLAN Not.* 50, 12 (Aug. 2015), 129–140.

[27] Nathanael Shärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. 2002. *Traits: Composable Units of Behavior*. Technical Report.

[28] Abraham Silberschatz, Henry Korth, and S. Sudarshan. 2006. *Database Systems Concepts* (5 ed.). McGraw-Hill, Inc., New York, NY, USA.

[29] Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. 2015. Injective Type Families for Haskell. *SIGPLAN Not.* 50, 12 (Aug. 2015), 118–128.

[30] Peter J. Stuckey and Martin Sulzmann. 2005. A Theory of Overloading. *ACM Trans. Program. Lang. Syst.* 27, 6 (Nov. 2005), 1216–1269.

[31] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI '07)*. ACM, New York, NY, USA, 53–66.

[32] Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. 2007. Understanding Functional Dependencies via Constraint Handling Rules. *J. Funct. Program.* 17, 1 (Jan. 2007), 83–129.

[33] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let Should Not Be Generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '10)*. ACM, NY, USA, 39–50.

[34] Dimitrios Vytiniotis, Simon Peyton jones, Tom Schrijvers, and Martin Sulzmann. 2011. Outsidein(x) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412.

[35] P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 60–76.

[36] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. ACM, New York, NY, USA, 53–66.

A Additional Judgments

In this section we present some additional judgments which we omitted from our main presentation.

A.1 Type Well-formedness

$\boxed{\Gamma \vdash_{\mathsf{ty}} \sigma}$ Type Well-formedness

$$\frac{a \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} a}\ \textsc{WfVar} \qquad \frac{}{\Gamma \vdash_{\mathsf{ty}} T}\ \textsc{WfCon}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \rho \qquad \Gamma \vdash_{\mathsf{ct}} Q}{\Gamma \vdash_{\mathsf{ty}} Q \Rightarrow \rho}\ \textsc{WfQual} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \tau_i}{\Gamma \vdash_{\mathsf{ty}} F(\overline{\tau})}\ \textsc{WfFam}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2}{\Gamma \vdash_{\mathsf{ty}} \tau_1\ \tau_2}\ \textsc{WfApp} \qquad \frac{\Gamma, a \vdash_{\mathsf{ty}} \sigma \qquad a \notin fv(\Gamma)}{\Gamma \vdash_{\mathsf{ty}} \forall a.\sigma}\ \textsc{WfAll}$$

$\boxed{\Gamma \vdash_{\mathsf{ct}} Q}$ Constraint Well-formedness

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2}{\Gamma \vdash_{\mathsf{ct}} \tau_1 \sim \tau_2}\ \textsc{WfEq} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \tau_i}{\Gamma \vdash_{\mathsf{ct}} TC\ \overline{\tau}}\ \textsc{WfCls}$$

A.2 Value Binding Typing

$\boxed{I;\Gamma_1 \vdash_{\mathsf{val}} val : \Gamma_2}$ Value Binding Typing

$$\frac{I;\Gamma \vdash_{\mathsf{tm}} e : \sigma}{I;\Gamma \vdash_{\mathsf{val}} (x = e) : [x : \sigma]}\ \textsc{Val}$$

A.3 Program Typing

$\boxed{\vdash_{\mathsf{pgm}} pgm}$ Program Typing

$$\frac{I = I_c, I_i \qquad \Gamma = \Gamma_c, \Gamma_v \qquad \overline{I;\Gamma \vdash_{\mathsf{cls}} cls : I_c;\Gamma_c} \qquad \overline{I_c;I_i;\Gamma \vdash_{\mathsf{inst}} inst : I_i} \qquad \overline{I;\Gamma \vdash_{\mathsf{val}} val : \Gamma_v}}{\vdash_{\mathsf{pgm}} \overline{cls}; \overline{inst}; \overline{val}}\ \textsc{Pgm}$$

A.4 Match Context Bindings

Throughout the paper we have denoted the evidence and typing bindings of a match context $\mathbb{E}$ as $P_{\mathbb{E}}$ and $\Gamma_{\mathbb{E}}$, respectively. Function $\textsc{binds}(\cdot)$ below illustrates how the bindings can be extracted from a match context:

$\boxed{\textsc{binds}(\mathbb{E}) = \Gamma; P}$ Bindings of Match Contexts

$$\begin{array}{ll}
\textsc{binds}(\Box) & = \bullet;\bullet \\
\textsc{binds}(\textbf{case}\ d\ \textbf{of}\ p \to \mathbb{E}) & = (\overline{b}, (\overline{f : \upsilon}), \Gamma); ((\overline{c : \psi}), (\overline{d : \tau}), P) \\
\quad \textbf{where}\quad p & \equiv K_{TC}\ \overline{b}\ \overline{(c : \psi)}\ \overline{(d : \tau)}\ \overline{(f : \upsilon)} \\
\qquad\qquad \Gamma; P & = \textsc{binds}(\mathbb{E})
\end{array}$$

A.5 Type Translation

$\boxed{\Gamma \vdash_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon}$ Elaboration of Types

$$\frac{a \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} a \rightsquigarrow a}\ \textsc{E\_TyVar} \qquad \frac{\Gamma, a \vdash_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon \qquad a \notin \Gamma}{\Gamma \vdash_{\mathsf{ty}} \forall a.\sigma \rightsquigarrow \forall a.\upsilon}\ \textsc{E\_TyAll}$$

$$\frac{\Gamma \vdash_{\mathsf{eq}} \phi_i \rightsquigarrow \psi_i \qquad \Gamma \vdash_{\mathsf{cc}} \pi_i \rightsquigarrow \upsilon_i \qquad \Gamma \vdash_{\mathsf{ty}} \tau \rightsquigarrow \upsilon}{\Gamma \vdash_{\mathsf{ty}} (\overline{\phi}, \overline{\pi}) \Rightarrow \tau \rightsquigarrow \overline{\psi} \Rightarrow \overline{\upsilon} \to \upsilon}\ \textsc{E\_TyQual}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2 \rightsquigarrow \upsilon_2}{\Gamma \vdash_{\mathsf{ty}} \tau_1\ \tau_2 \rightsquigarrow \upsilon_1\ \upsilon_2}\ \textsc{E\_TyApp}$$

$$\frac{}{\Gamma \vdash_{\mathsf{ty}} T \rightsquigarrow T}\ \textsc{E\_TyCon} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \tau_i \rightsquigarrow \upsilon_i}{\Gamma \vdash_{\mathsf{ty}} F(\overline{\tau}) \rightsquigarrow F(\overline{\upsilon})}\ \textsc{E\_TyFam}$$

A.6 Constraint Translation

$\boxed{\Gamma \vdash_{\mathsf{cc}} \pi \rightsquigarrow \upsilon}$ Elaboration of Class Constraints

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_i \rightsquigarrow \upsilon_i}{\Gamma \vdash_{\mathsf{cc}} TC\ \overline{\tau} \rightsquigarrow T_{TC}\ \overline{\upsilon}}\ \textsc{E\_ClsCt}$$

$\boxed{\Gamma \vdash_{\mathsf{eq}} \phi \rightsquigarrow \psi}$ Elaboration of Equality Constraints

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \rightsquigarrow \upsilon_1 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_2 \rightsquigarrow \upsilon_2}{\Gamma \vdash_{\mathsf{eq}} (\tau_1 \sim \tau_2) \rightsquigarrow (\upsilon_1 \sim \upsilon_2)}\ \textsc{E\_EqCt}$$

A.7 Constraint Entailment

$\boxed{P \vdash_{\mathsf{E}} C \rightsquigarrow^{*} C; \theta; \eta}$ Constraint Entailment

$$\frac{}{P \vdash_{\mathsf{E}} C \rightsquigarrow^{*} C; \bullet; \bullet}\ \textsc{Stop}_E$$

$$\frac{P \vdash_{\mathsf{E}} Q \rightsquigarrow C_1; \theta_1; \eta_1 \qquad P \vdash_{\mathsf{E}} \theta_1(C), C_1 \rightsquigarrow^{*} C_2; \theta_2; \eta_2}{P \vdash_{\mathsf{E}} C, Q \rightsquigarrow^{*} C_2; (\theta_2 \cdot \theta_1); (\eta_2 \cdot \eta_1)}\ \textsc{Step}_E$$

A.8 Value Binding Translation

Relation $P;\Gamma \vdash_{\mathsf{val}} val : \Gamma_{\upsilon} \rightsquigarrow decl$ performs type inference (and elaboration into System F_C) for top-level bindings.

$\boxed{P;\Gamma \vdash_{\mathsf{val}} val : \Gamma_{\upsilon} \rightsquigarrow decl}$ Value Binding Elaboration

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{tm}} e : \tau \rightsquigarrow t \mid \mathcal{P}_1; \mathcal{E}_1 \\ P \vdash_{\mathsf{E}} (\mathcal{P}_1, \mathcal{E}_1) \rightsquigarrow^{!} (\mathcal{P}_2, \mathcal{E}_2); \theta; \eta \qquad \overline{a} = fuv(\mathcal{P}_2, \mathcal{E}_2, \theta(\tau)) \\ \sigma = \forall \overline{a}.(erase(\mathcal{E}), erase(\mathcal{P})) \Rightarrow \theta(\tau) \qquad \Gamma \vdash_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon\end{array}}{P;\Gamma \vdash_{\mathsf{val}} (x = e) : [x : \sigma] \rightsquigarrow \textbf{let}\ x : \upsilon = \Lambda\overline{a}.\Lambda\mathcal{E}_2.\lambda\mathcal{P}_2.\theta(\eta(t))}\ \textsc{E\_Val}$$

To stay in line with the primary goal of functional dependencies, type improvement [14], once we have inferred the type of the binding we perform simplification, via constraint entailment (function $erase(\cdot)$ removes all evidence annotations from either class constraints $\mathcal{P}$ or equality constraints $\mathcal{E}$).

Notice that, for simplicity, we have considered only top-level bindings without type signatures but it is extremely straightforward to allow explicit type annotations. A top-level binding $val \equiv (x :: \sigma) = e$ can be handled by relation $P;\Gamma \vdash_{\mathsf{tm}} e : \sigma \rightsquigarrow t$, given in Section 5.6.2:

$$\frac{P;\Gamma \vdash_{\mathsf{tm}} e : \sigma \rightsquigarrow t \qquad \Gamma \vdash_{\mathsf{ty}} \sigma \rightsquigarrow \upsilon}{P;\Gamma \vdash_{\mathsf{val}} (x :: \sigma = e) : \Gamma_{\upsilon} \rightsquigarrow \textbf{let}\ x : \upsilon = t}\ \textsc{E\_ValAnn}$$

A.9 Program Translation

$\boxed{\vdash_{\mathsf{pgm}} pgm \rightsquigarrow \overline{decl}}$ Program Elaboration

$$\frac{\Gamma = \Gamma_c, \Gamma_{\upsilon} \qquad \overline{\Gamma \vdash_{\mathsf{cls}} cls \rightsquigarrow decl_c \mid \Gamma_c} \qquad \overline{P;\Gamma \vdash_{\mathsf{inst}} inst \rightsquigarrow decl_i \mid P} \qquad \overline{P;\Gamma \vdash_{\mathsf{val}} val : \Gamma_{\upsilon} \rightsquigarrow decl_{\upsilon}}}{\vdash_{\mathsf{pgm}} \overline{cls}; \overline{inst}; \overline{val} \rightsquigarrow \overline{decl_c}; \overline{decl_i}; \overline{decl_{\upsilon}}}\ \textsc{E\_Pgm}$$

A.10 Poly-kinded, Generic Type Projections

Even though we omitted kinds from our main presentation for brevity, it is quite straightforward to extend the system of Section 4 with kind checking (quite cumbersome though).

More importantly, if we further extend the system with *kind polymorpism* [36] – which is also straightforward – there is no need to axiomatize the projection functions we presented in Section 4.1.4 for each data type.

Instead, we can perform type projection generically, using only two user-defined kind-polymorphic type families L and R, along with two axioms:

$$\begin{array}{lcl} \textbf{type}\ L & : & \forall \kappa_1\ \kappa_2.\ (a : \kappa_1) \rightarrow \kappa_2 \\ \textbf{type}\ R & : & \forall \kappa_1\ \kappa_2.\ (a : \kappa_1) \rightarrow \kappa_2 \end{array}$$

$$\begin{array}{lcl} \textbf{axiom}\ proj_L & : & L\ ((u_1 : \kappa_2 \rightarrow \kappa_1)\ (u_2 : \kappa_2)) \sim u_1 \\ \textbf{axiom}\ proj_R & : & R\ ((u_1 : \kappa_2 \rightarrow \kappa_1)\ (u_2 : \kappa_2)) \sim u_2 \end{array}$$

For example, instead of the axioms $Fst\ (a, b) \sim a$ and $Snd\ (a, b) \sim b$, we can extract the first and the second component of a tuple type τ as follows:

$$\begin{array}{lcl} Fst\ \tau & \equiv & R\ (L\ \tau) \\ Snd\ \tau & \equiv & R\ \tau \end{array}$$

B System $\mathrm{F_C}$ Type System

B.1 Coercion Typing

$\boxed{P; \Gamma \vdash_{\mathsf{co}} \gamma : \psi}$ Coercion Typing

$$\frac{(c : \psi) \in P}{P; \Gamma \vdash_{\mathsf{co}} c : \psi}\ \textsc{CoVar} \qquad \frac{(g\ \overline{a} : \upsilon_1 \sim \upsilon_2) \in P \qquad \overline{\Gamma \vdash_{\mathsf{ty}} \upsilon}}{P; \Gamma \vdash_{\mathsf{co}} g\ \overline{\upsilon} : [\overline{\upsilon}/\overline{a}]\upsilon_1 \sim [\overline{\upsilon}/\overline{a}]\upsilon_2}\ \textsc{CoAx}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \upsilon}{P; \Gamma \vdash_{\mathsf{co}} \langle \upsilon \rangle : \upsilon \sim \upsilon}\ \textsc{CoRefl} \qquad \frac{P; \Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2}{P; \Gamma \vdash_{\mathsf{co}} \mathsf{sym}\ \gamma : \upsilon_2 \sim \upsilon_1}\ \textsc{CoSym}$$

$$\frac{P; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \upsilon_1 \sim \upsilon_2 \qquad P; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \upsilon_2 \sim \upsilon_3}{P; \Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \upsilon_1 \sim \upsilon_3}\ \textsc{CoTrans}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \upsilon_1\ \upsilon_3 \qquad P; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \upsilon_1 \sim \upsilon_2 \qquad P; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \upsilon_3 \sim \upsilon_4}{P; \Gamma \vdash_{\mathsf{co}} \gamma_1\ \gamma_2 : \upsilon_1\ \upsilon_3 \sim \upsilon_2\ \upsilon_4}\ \textsc{CoApp}$$

$$\frac{P; \Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1\ \upsilon_2 \sim \upsilon_3\ \upsilon_4}{P; \Gamma \vdash_{\mathsf{co}} \mathsf{left}\ \gamma : \upsilon_1 \sim \upsilon_3}\ \textsc{CoL} \qquad \frac{P; \Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1\ \upsilon_2 \sim \upsilon_3\ \upsilon_4}{P; \Gamma \vdash_{\mathsf{co}} \mathsf{right}\ \gamma : \upsilon_2 \sim \upsilon_4}\ \textsc{CoR}$$

$$\frac{F_n \in \Gamma \qquad \overline{P; \Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2}^{n} \qquad \overline{\Gamma \vdash_{\mathsf{ty}} \upsilon_1}^{n}}{P; \Gamma \vdash_{\mathsf{co}} F_n(\overline{\gamma}^{n}) : F(\overline{\upsilon_1}^{n}) \sim F(\overline{\upsilon_2}^{n})}\ \textsc{CoFam}$$

$$\frac{P; \Gamma, a \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2 \qquad \Gamma, a \vdash_{\mathsf{ty}} \upsilon_1 \qquad a \notin \Gamma}{P; \Gamma \vdash_{\mathsf{co}} \forall a.\gamma : \forall a.\upsilon_1 \sim \forall a.\upsilon_2}\ \textsc{CoAll}$$

$$\frac{\begin{array}{c} P; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \forall a.\upsilon_1 \sim \forall a.\upsilon_2 \\ P; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \upsilon_3 \sim \upsilon_4 \qquad \Gamma \vdash_{\mathsf{ty}} \upsilon_3 \end{array}}{P; \Gamma \vdash_{\mathsf{co}} \gamma_1[\gamma_2] : [\upsilon_3/a]\upsilon_1 \sim [\upsilon_4/a]\upsilon_2}\ \textsc{CoInst}$$

$$\frac{\Gamma \vdash_{\mathsf{pr}} \psi \qquad P; \Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2}{P; \Gamma \vdash_{\mathsf{co}} \psi \Rightarrow \gamma : (\psi \Rightarrow \upsilon_1) \sim (\psi \Rightarrow \upsilon_2)}\ \textsc{CoQual}$$

$$\frac{P; \Gamma \vdash_{\mathsf{co}} \gamma_1 : (\psi \Rightarrow \upsilon_1) \sim (\psi \Rightarrow \upsilon_2) \qquad P; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \psi}{P; \Gamma \vdash_{\mathsf{co}} \gamma_1 @ \gamma_2 : \upsilon_1 \sim \upsilon_2}\ \textsc{CoQInst}$$

B.2 Type Well-formedness

$\boxed{\Gamma \vdash_{\mathsf{ty}} \upsilon}$ Type Well-formedness

$$\frac{a \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} a}\ \textsc{TyVar} \qquad \frac{\Gamma \vdash_{\mathsf{pr}} \psi \qquad \Gamma \vdash_{\mathsf{ty}} \upsilon}{\Gamma \vdash_{\mathsf{ty}} \psi \Rightarrow \upsilon}\ \textsc{TyQual}$$

$$\frac{T \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} T}\ \textsc{TyCon} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \upsilon_1 \qquad \Gamma \vdash_{\mathsf{ty}} \upsilon_2}{\Gamma \vdash_{\mathsf{ty}} \upsilon_1\ \upsilon_2}\ \textsc{TyApp}$$

$$\frac{\Gamma, a \vdash_{\mathsf{ty}} \upsilon \qquad a \notin fv(\Gamma)}{\Gamma \vdash_{\mathsf{ty}} \forall a.\upsilon}\ \textsc{TyAll} \qquad \frac{\Gamma \vdash_{\mathsf{ty}} \upsilon_i}{\Gamma \vdash_{\mathsf{ty}} F_{TC_i}(\overline{\upsilon})}\ \textsc{TyFam}$$

$\boxed{\Gamma \vdash_{\mathsf{pr}} \psi}$ Proposition Well-formedness

$$\frac{\Gamma \vdash_{\mathsf{ty}} \upsilon_1 \qquad \Gamma \vdash_{\mathsf{ty}} \upsilon_2}{\Gamma \vdash_{\mathsf{pr}} \upsilon_1 \sim \upsilon_2}\ \textsc{Prop}$$

B.3 Term Typing

$\boxed{P; \Gamma \vdash_{\mathsf{tm}} t : \upsilon}$ Term Typing

$$\frac{(x : \upsilon) \in \Gamma}{P; \Gamma \vdash_{\mathsf{tm}} x : \upsilon}\ \textsc{TmVar} \qquad \frac{(d : \tau) \in P}{P; \Gamma \vdash_{\mathsf{tm}} d : \upsilon}\ \textsc{TmD} \qquad \frac{(K : \upsilon) \in \Gamma}{P; \Gamma \vdash_{\mathsf{tm}} K : \upsilon}\ \textsc{TmCon}$$

$$\frac{P; \Gamma \vdash_{\mathsf{tm}} t : \upsilon_1 \qquad P; \Gamma \vdash_{\mathsf{co}} \gamma : \upsilon_1 \sim \upsilon_2}{P; \Gamma \vdash_{\mathsf{tm}} t \triangleright \gamma : \upsilon_2}\ \textsc{TmCast}$$

$$\frac{a \notin \Gamma \qquad P; \Gamma, a \vdash_{\mathsf{tm}} t : \upsilon}{P; \Gamma \vdash_{\mathsf{tm}} \Lambda a.t : \forall a.\upsilon}\ (\forall I) \qquad \frac{\begin{array}{c} P; \Gamma \vdash_{\mathsf{tm}} t : \forall a.\upsilon \\ \Gamma \vdash_{\mathsf{ty}} \upsilon_1 \end{array}}{P; \Gamma \vdash_{\mathsf{tm}} t\ \upsilon : [\upsilon_1/a]\upsilon}\ (\forall E)$$

$$\frac{P; \Gamma \vdash_{\mathsf{tm}} t_1 : \upsilon_2 \rightarrow \upsilon_1 \qquad P; \Gamma \vdash_{\mathsf{tm}} t_2 : \upsilon_2}{P; \Gamma \vdash_{\mathsf{tm}} t_1\ t_2 : \upsilon_1}\ (\rightarrow E)$$

$$\frac{x \notin dom(\Gamma) \qquad \Gamma \vdash_{\mathsf{ty}} \upsilon_1 \qquad P; \Gamma, x : \upsilon_1 \vdash_{\mathsf{tm}} t : \upsilon_2}{P; \Gamma \vdash_{\mathsf{tm}} \lambda(x : \upsilon_1).t : \upsilon_1 \rightarrow \upsilon_2}\ (\rightarrow I)$$

$$\frac{\begin{array}{c} c \notin dom(P) \qquad \Gamma \vdash_{\mathsf{pr}} \psi \\ P; \Gamma, c : \psi \vdash_{\mathsf{tm}} t : \upsilon \end{array}}{P; \Gamma \vdash_{\mathsf{tm}} \Lambda(c : \psi).t : \psi \Rightarrow \upsilon}\ (\Rightarrow I_{\psi}) \qquad \frac{\begin{array}{c} P; \Gamma \vdash_{\mathsf{tm}} t : \psi \Rightarrow \upsilon \\ P; \Gamma \vdash_{\mathsf{co}} \gamma : \psi \end{array}}{P; \Gamma \vdash_{\mathsf{tm}} t\ \gamma : \upsilon}\ (\Rightarrow E_{\psi})$$

$$\frac{\begin{array}{c} x \notin dom(\Gamma) \\ P; \Gamma, x : \upsilon_1 \vdash_{\mathsf{tm}} t_1 : \upsilon_1 \qquad \Gamma \vdash_{\mathsf{ty}} \upsilon_1 \qquad P; \Gamma, x : \upsilon_1 \vdash_{\mathsf{tm}} t_2 : \upsilon_2 \end{array}}{P; \Gamma \vdash_{\mathsf{tm}} (\textbf{let}\ x : \upsilon_1 = t_1\ \textbf{in}\ t_2) : \upsilon_2}\ \textsc{TmLet}$$

$$\frac{P; \Gamma \vdash_{\mathsf{tm}} t_1 : \upsilon_1 \qquad P; \Gamma \vdash_{\mathsf{p}} p \rightarrow t_2 : \upsilon_1 \rightarrow \upsilon_2}{P; \Gamma \vdash_{\mathsf{tm}} \textbf{case}\ t_1\ \textbf{of}\ p \rightarrow t_2 : \upsilon_2}\ \textsc{TmCase}$$

$\boxed{P; \Gamma \vdash_{\mathsf{p}} p \rightarrow t : \upsilon_1 \rightarrow \upsilon_2}$ Pattern Typing

$$\frac{\begin{array}{c} (K : \forall \overline{a}\overline{b}'.\overline{\psi} \Rightarrow \overline{\tau} \rightarrow \overline{\upsilon} \rightarrow T\ \overline{a}) \in \Gamma \qquad \theta = [\overline{\tau}_a/\overline{a}, \overline{b}'/\overline{b}] \\ \overline{b} \notin \Gamma \qquad \overline{c}, \overline{d} \notin dom(P) \qquad \overline{x} \notin dom(\Gamma) \\ P, \overline{(c : \theta(\psi))}, \overline{(d : \theta(\tau))}; \Gamma, \overline{b}, \overline{(x : \theta(\upsilon))} \vdash_{\mathsf{tm}} t : \upsilon_2 \end{array}}{P; \Gamma \vdash_{\mathsf{p}} K\ \overline{b}\ \overline{(c : \theta(\psi))}\ \overline{(d : \theta(\tau))}\ \overline{(x : \theta(\upsilon))} \rightarrow t : T\ \overline{\tau}_a \rightarrow \upsilon_2}\ \textsc{Pat}$$

B.4 Declaration & Program Typing

$\boxed{P_1; \Gamma_1 \vdash_{\mathsf{d}} decl : P_2; \Gamma_2}$ Declaration Typing

$$\frac{\Gamma, \overline{a}, \overline{b} \vdash_{\mathsf{pr}} \psi_i \qquad \Gamma, \overline{a}, \overline{b} \vdash_{\mathsf{ty}} \upsilon_j}{P; \Gamma \vdash_{\mathsf{d}} (\textbf{data}\ T\ \overline{a}\ \textbf{where}\ K : \forall \overline{a}\overline{b}.\overline{\psi} \Rightarrow \overline{\upsilon} \rightarrow T\ \overline{a}) : \bullet; \bullet}\ \textsc{Data}$$

$$\frac{}{P; \Gamma \vdash_{\mathsf{d}} \textbf{type}\ F(\overline{a}) : \bullet; \bullet}\ \textsc{Family}$$

$$\frac{\Gamma, \overline{a} \vdash_{\mathsf{ty}} u_i \qquad \Gamma, \overline{a} \vdash_{\mathsf{ty}} \upsilon \qquad g \notin dom(P)}{P; \Gamma \vdash_{\mathsf{d}} (\textbf{axiom}\ g\ \overline{a} : F(\overline{u}) \sim \upsilon) : [g\ \overline{a} : F(\overline{u}) \sim \upsilon]; \bullet}\ \textsc{Axiom}$$

$$\frac{P; \Gamma, x : \upsilon \vdash_{\mathsf{tm}} t : \upsilon \qquad x \notin dom(\Gamma)}{P; \Gamma \vdash_{\mathsf{d}} (\textbf{let}\ x : \upsilon = t) : \bullet; [x : \upsilon]}\ \textsc{Value}$$

$\boxed{\vdash^{\mathsf{fc}}_{\mathsf{pgm}} \overline{decl}}$ Program Typing

$$\frac{\overline{P; \Gamma \vdash_{\mathsf{d}} decl : P; \Gamma}}{\vdash^{\mathsf{fc}}_{\mathsf{pgm}} \overline{decl}}\ \textsc{Program}$$

C Constraint Schemes, CHRs and System F_C

In this section we informally illustrate that both our specification (Section 4) and our elaboration (Section 5) semantics are compatible with the Constraint Handling Rules of Sulzmann et al. [32].

C.1 Class CHRs

Let there be a class declaration

$$\textbf{class}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{a} \mid fd_1, \dots, fd_m$$

According to Sulzmann et al., it gives rise to two kinds of constraint handling rules:

Class CHR. The class chr takes the form $\textbf{rule}\ TC\ \overline{a} \implies \overline{\pi}$, that is, almost the same as Scheme CS1a:

$$S_{C_\pi} = \forall \overline{a}.TC\ \overline{a} \Rightarrow \theta(\pi) \quad \forall \pi$$

where $\theta = det(\overline{a}, \overline{\pi})$. The main difference between the two is the substitution θ, which essentially replaces all skolem variables $\overline{b}$ in $\overline{\pi}$ with their (known) counterparts.[8]

The CHR has a direct interpretation into System F_C, as a match context:

$$\mathbb{E} = \textbf{case}\ (d : T_{TC}\ \overline{a})\ \textbf{of}\ \{\ \dots\ \}$$

Within the scope of $\mathbb{E}$, both $\overline{b}$ and $\overline{\pi}$ are available. The elaboration of the constraint scheme is slightly more complex. Matching against all superclass constraints $\overline{\pi}$ recursively makes available all coercions that connect $\overline{b}$ with $\overline{a}$. Composed, they can be used to cast the type of the superclass constraints to $\theta(\pi)$. As an example, consider the following definitions:

$$\begin{array}{l}\textbf{class}\ C\ a\ b \mid a \to b \\ \textbf{class}\ C\ a\ b \Rightarrow D\ a\end{array}$$

The corresponding constraint scheme is

$$\forall a.D\ a \Rightarrow C\ a\ (F_C\ a)$$

and is witnessed by the following function:

$$\begin{array}{l} f = \Lambda a.\lambda(d_1 : T_D\ a).\textbf{case}\ d_1\ \textbf{of} \\ \quad T_D\ b\ (d_2 : T_C\ a\ b) \to \textbf{case}\ d_2\ \textbf{of} \\ \quad\quad T_C\ (c : F_C\ a \sim b) \to d_2 \triangleright \langle T_C \rangle\ \langle a \rangle\ (\text{sym}\ c) \end{array}$$

This explains why we strictly require that $\overline{b} \subseteq dom(\theta)$: if the restriction does not hold, there are superclass constraint schemes that cannot be elaborated in System F_C. Such declarations are ambiguous, so we consider it a reasonable restriction.

Functional Dependency CHRs. For each functional dependency $fd_i \equiv a_{i_1} \dots a_{i_n} \to a_{i_0}$, we have $\textbf{rule}\ TC\ \overline{a}, TC\ \theta(\overline{b}) \implies a_{i_0} \sim b_{i_0}$, where

$$\theta(b_j) = \begin{cases} a_j & \text{, if } j \in \{i_1, \dots, i_n\} \\ b_j & \text{, otherwise} \end{cases}$$

This rule is derivable using Scheme CS1b twice as follows:

$$\dfrac{\dfrac{\dfrac{TC\ \overline{a}}{F_{TC_i}\ \overline{a}^{i_n} \sim a_{i_0}}\ \textsc{1b}}{a_{i_0} \sim F_{TC_i}\ \overline{a}^{i_n}}\ \textsc{Sym} \qquad \dfrac{\dfrac{TC\ \theta(\overline{b})}{F_{TC_i}\ \theta(\overline{b}^{i_n}) \sim \theta(b_{i_0})}\ \textsc{1b}}{F_{TC_i}\ \overline{a}^{i_n} \sim b_{i_0}}\ \theta}{a_{i_0} \sim b_{i_0}}\ \fatsemi$$

[8]If we restrict ourselves to cases where $\overline{b} \subseteq dom(\theta)$, both our specification and the inference (with elaboration) are well-behaved.

The System F_C counterpart of this constraint scheme is the combination of two match contexts

$$\begin{array}{l} \mathbb{E} = \textbf{case}\ (d_1 : T_{TC}\ \overline{a})\ \textbf{of} \\ \quad T_{TC}\ \dots\ \overline{c}_1\ \dots \to \textbf{case}\ (d_2 : T_{TC}\ \theta(\overline{b}))\ \textbf{of} \\ \quad\quad T_{TC}\ \dots\ \overline{c}_2\ \dots \to \square \end{array}$$

and a *local* coercion $\gamma = (\text{sym}\ c_{i1}) \fatsemi c_{i2}$. Notice the importance of the match context, for the well-scopedness of the coercion: sub-coercions c_{i1} and c_{i2} are available only within the scope of the match context.

C.2 Instance CHRs

Let there be an instance declaration

$$\textbf{instance}\ \forall \overline{a}\overline{b}.\overline{\pi} \Rightarrow TC\ \overline{u}$$

According to Sulzmann et al., it also gives rise to two kinds of constraint handling rules:

Instance CHR. The instance rule takes the form $\textbf{rule}\ TC\ \overline{u} \iff \overline{\pi}$, which, according to CHR semantics, means that instead of proving $TC\ \overline{u}$, one suffices to prove $\overline{\pi}$. That is, we can always derive $TC\ \overline{u}$ from $\overline{\pi}$. This directly corresponds to Scheme CS2a:

$$S_{I_{\overline{\pi}}} = \forall \overline{a}.\theta(\overline{\pi}) \Rightarrow TC\ \overline{u}$$

where $\theta = det(\overline{a}, \overline{\pi})$. The interpretation of this scheme is the expected dictionary constructor, where $\overline{b}$ are appropriately instantiated. For a system that does not support the *type substitution property*, this wouldn't necessarily be accepted, since the quantification over $\overline{b}$ is non-parametric. Yet, as we illustrated in Section 6.3, our system does, and our instantiation is (by construction) the expected.

Instance Improvement CHRs. For each functional dependency $fd_i \equiv a_{i_1} \dots a_{i_n} \to a_{i_0}$, we have $\textbf{rule}\ TC\ \theta'(\overline{b}) \implies u_{i_0} \sim b_{i_0}$, where

$$\theta'(b_j) = \begin{cases} u_j & \text{, if } j \in \{i_1, \dots, i_n\} \\ b_j & \text{, otherwise} \end{cases}$$

We can derive this rule, by combining Schemes CS1b and CS2b:

$$\dfrac{\dfrac{\dfrac{}{F_{TC_i}\ \overline{u}^{i_n} \sim \theta(u_{i_0})}\ \textsc{2b}}{\theta(u_{i_0}) \sim F_{TC_i}\ \overline{u}^{i_n}}\ \textsc{Sym} \qquad \dfrac{\dfrac{TC\ \theta'(\overline{b})}{F_{TC_i}\ \theta'(\overline{b}^{i_n}) \sim \theta'(b_{i_0})}\ \textsc{1b}}{F_{TC_i}\ \overline{u}^{i_n} \sim b_{i_0}}\ \theta'}{\theta(u_{i_0}) \sim b_{i_0}}\ \fatsemi$$

The corresponding System F_C term has exactly the same structure: a local pattern match (encoding Scheme CS1b) against the dictionary of type $(TC\ \theta'(\overline{b}))$ in order to expose equality $F_{TC_i}\ \overline{u}^{i_n} \sim b_{i_0}$, which is then combined with top-level axiom $\theta(u_{i_0}) \sim F_{TC_i}\ \overline{u}^{i_n}$ (from Scheme CS2b) to produce $\theta(u_{i_0}) \sim b_{i_0}$.

Notice that similarly to the Instance CHR, we do not actually prove $u_{i_0} \sim b_{i_0}$, but the refined $\theta(u_{i_0}) \sim b_{i_0}$.