Coherence of Type Class Resolution

GERT-JAN BOTTU, KU Leuven, Belgium
NINGNING XIE, The University of Hong Kong, China
KOAR MARNTIROSIAN, KU Leuven, Belgium
TOM SCHRIJVERS, KU Leuven, Belgium

Elaboration-based type class resolution, as found in languages like Haskell, Mercury and PureScript, is generally nondeterministic: there can be multiple ways to satisfy a wanted constraint in terms of global instances and locally given constraints. Coherence is the key property that keeps this sane; it guarantees that, despite the nondeterminism, programs still behave predictably. Even though elaboration-based resolution is generally assumed coherent, as far as we know, there is no formal proof of this property in the presence of sources of nondeterminism, like superclasses and flexible contexts.

This paper provides a formal proof to remedy the situation. The proof is non-trivial because the semantics elaborates resolution into a target language where different elaborations can be distinguished by contexts that do not have a source language counterpart. Inspired by the notion of full abstraction, we present a two-step strategy that first elaborates nondeterministically into an intermediate language that preserves contextual equivalence, and then deterministically elaborates from there into the target language. We use an approach based on logical relations to establish contextual equivalence and thus coherence for the first step of elaboration, while the second step's determinism straightforwardly preserves this coherence property.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → Correctness**; **Functional languages**.

Additional Key Words and Phrases: type class resolution, coherence, logical relations

ACM Reference Format:
Gert-Jan Bottu, Ningning Xie, Koar Marntirosian, and Tom Schrijvers. 2019. Coherence of Type Class Resolution. *Proc. ACM Program. Lang.* 3, ICFP, Article 91 (August 2019), 28 pages. https://doi.org/10.1145/3341695

As a guide to the reader, we present the source language $\lambda_{\mathbf{TC}}$ in blue, the intermediate language $F_{\mathbf{D}}$ in green and the target language $F_{\{\}}$ in red. We thus encourage the reader to view / print this paper in color.

1 INTRODUCTION

Type classes were initially introduced in Haskell [Peyton Jones 2003] by Wadler and Blott [Wadler and Blott 1989] to make ad-hoc overloading less ad hoc, and they have since become one of Haskell's core abstraction features. Moreover, their resounding success has spread far beyond Haskell: several languages have adopted them (e.g., Mercury [Henderson et al. 1996], Coq [Sozeau and Oury 2008], PureScript [Freeman 2017], Lean [de Moura et al. 2015]), and they have inspired various alternative language features (e.g., Scala's implicits [Martin Odersky and Venners 2008;

Authors' addresses: Gert-Jan Bottu, Department of Computer Science, KU Leuven, Belgium, gertjan.bottu@kuleuven.be; Ningning Xie, Department of Computer Science, The University of Hong Kong, China, nnxie@cs.hku.hk; Koar Marntirosian, Department of Computer Science, KU Leuven, Belgium, koar.marntirosian@kuleuven.be; Tom Schrijvers, Department of Computer Science, KU Leuven, Belgium, tom.schrijvers@kuleuven.be.

2475-1421/2019/8-ART91
https://doi.org/10.1145/3341695

Odersky et al. 2017], Rust's traits [Mozilla Research 2017], C++'s concepts [Gregor et al. 2006], Agda's instance arguments [Devriese and Piessens 2011]).

Type classes have also received a lot of attention from researchers with many proposals for extensions and improvements, including functional dependencies [Jones 2000], associated types [Chakravarty et al. 2005], quantified constraints [Bottu et al. 2017] among other extensions.

Given the extensive attention that type classes have received, it may be surprising that the metatheory of their elaboration-based semantics [Hall et al. 1996] has not yet been exhaustively studied. In particular, as far as we know, while there have been many informal arguments, the formal notion of *coherence* has never been proven. Reynolds [1991] has defined coherence as follows:

> *"When a programming language has a sufficiently rich type structure, there can be more than one proof of the same typing judgment; potentially this can lead to semantic ambiguity since the semantics of a typed language is a function of such proofs. When no such ambiguity arises, we say that the language is coherent."*

Type classes give rise to two main (potential) sources of incoherence. The first source are *ambiguous type schemes*, such as that of the function `foo`:

```
> foo :: (Show a, Read a) => String -> String
> foo s = show s (read s)
```

The type scheme of `foo` requires that the type with which `a` will be instantiated must have `Show` and `Read` instances. This restriction alone is too permissive, because the type part (`String -> String`) of `foo`'s type scheme is not sufficient for a deterministic instantiation of `a` from the calling context. `a` can thus be instantiated arbitrarily to any type with `Show` and `Read` instances. Yet, the choice of type may lead to a different behavior of `show` and `read`, and thus of `foo` as a whole. For instance, `foo "1"` yields `"1"` when `a` is instantiated to `Int`, and `"1.0"` when it is instantiated to `Float`. To rule out this source of incoherence, Jones [1993] requires type schemes to be unambiguous and has formally proven that, for his system, this guarantees coherence.

The second source of ambiguity arises from the type class resolution mechanism itself. Such mechanisms check whether a particular type class constraint holds. Usually, they are styled after resolution-based proof search in logic, where type class instances act as Horn clauses and type scheme constraints as additional facts. Generally, this process is nondeterministic, but languages like Haskell, Mercury or PureScript contain it by requiring that type class instances do not overlap with each other or with locally given constraints. Nevertheless, superclasses remain as a source of nondeterminism; indeed, a superclass constraint can be resolved through any of its subclass constraints. Hence, in the presence of superclasses, type class resolution should properly be considered as a potential source for incoherence. Moreover, overlap between locally wanted constraints and global instances is often allowed (e.g., through GHC's `FlexibleContexts` pragma), but a formal argument for its harmlessness is also lacking. Jones [1993] considered neither of these aspects and simply assumed the coherence of resolution as a given. Morris [2014] side-steps these issues with a denotational semantics that is disconnected from the original elaboration-based semantics and its implementations (e.g., Hugs and GHC).

This paper aims to fill this gap in the metatheory of programming languages featuring type classes, including industrial grade languages such as Haskell, by formally establishing that elaboration-based type class resolution is coherent in the presence of superclasses and flexible contexts. The proof of this property is considerably complicated by the indirect, elaboration-based approach that is used to give meaning to programs with type classes. Indeed, the meaning of such programs is commonly given in terms of their translation to a core language [Hall et al. 1996], like System F, the meaning of which is defined in the form of an operational semantics. In this translation process, type classes are elaborated into explicitly passed function dictionaries. These dictionaries

can, however, often be constructed in more than one way, resulting in multiple possible translations for a single program. The problem is that different translations of the same source program actually may have different meanings in the core language. The reason for this discrepancy is that the core language is more expressive than the source language and admits programs — that cannot be expressed in the source language — in which the different dictionaries can be distinguished.

We solve this problem with a new two-step approach that splits the problem into two subproblems. The midway point is an intermediate language that makes type class dictionaries explicit, but—inspired by fully abstract compilation—cannot distinguish between different elaborations from the same source language term [Abadi 1999]. We use a logical-relations approach to show that the nondeterministic elaboration from the source language to this intermediate language is coherent. Showing coherence for the elaboration from the intermediate language to the target language is much simpler, because we can formulate it in a deterministic fashion.

In summary, the contributions of this work are:

- We present a simple calculus $\lambda_{\mathbf{TC}}$ with full-blown type class resolution (incl. superclasses), which isolates nondeterministic resolution. Furthermore, we present an elaboration from $\lambda_{\mathbf{TC}}$ to the target language $F_{\{\}}$, System F with records, which are used to encode dictionaries.
- We present an intermediate language $F_{\mathbf{D}}$ with explicit dictionary-passing. This language enforces the uniqueness of dictionaries, which captures the intention of type class instances. We study its metatheory, and define a logical relation to prove contextual equivalence.
- We present elaborations from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$ and from $F_{\mathbf{D}}$ to $F_{\{\}}$, and prove that a direct translation from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$ can always be decomposed into an equivalent translation through $F_{\mathbf{D}}$.
- We prove coherence of the elaboration between $\lambda_{\mathbf{TC}}$ and $F_{\mathbf{D}}$, using logical relations.
- We prove that coherence is also preserved through the elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$. As a consequence, by combining this with the previous result, we prove that the elaboration between $\lambda_{\mathbf{TC}}$ and $F_{\{\}}$ is coherent. The latter coherence result implies coherence of elaboration-based type class resolution in the presence of superclasses and flexible contexts.

The full formalization and coherence proof can be found in the accompanying 147-page appendix.

The purpose of our work is twofold: 1) To develop a proof technique to establish coherence of type class resolution. Because this result is achieved on a minimal calculus, this work becomes a basis for researchers investigating type class extensions and larger languages, as well as their impact on coherence. 2) To present a formal proof of coherence for language designers considering to adopt type classes. In doing so, we show that the informally trivial argument for the coherence of type class resolution is surprisingly hard to formalize.

2 OVERVIEW

This section provides some background on dictionary-passing elaboration of type class resolution and discusses the potential nondeterminism introduced by superclasses and local constraints. We then briefly introduce our calculi and discuss the key ideas of the coherence proof. Throughout the section we use Haskell-like syntax as the source language for examples, and to simplify our informal discussion we use the same syntax without type classes as the target language.

2.1 Dictionary-Passing Elaboration

A program is coherent if it has exactly one meaning — i.e., its semantics is unambiguously determined. For type classes this is not as straightforward as it seems, because their dynamic semantics are not expressed directly but rather by type-directed elaboration into a simpler language without type classes such as System F. Thus the dynamic semantics of type classes are given indirectly as the dynamic semantics of their elaborated forms.

```
> class Eq a where
>   (==) :: a -> a -> Bool
>
> instance Eq Int where
>   (==) = primEqInt
>
> instance (Eq a, Eq b) => Eq (a, b) where
>   (x1,y1) == (x2,y2)  =  x1 == x2 && y1 == y2
>
> refl :: Eq a => a -> Bool
> refl x = x == x
>
> main :: Bool
> main = refl (5,42)
```

Example 1. Program with type classes.

Basic Elaboration. Consider the small program with type classes in Example 1. We declare a type class `Eq` and instances for the `Int` and pair types. The function `refl` trivially tests whether an expression is equivalent to itself, which is called in `main`.

The dictionary-passing elaboration translates this program into a System F-like core language that does not feature type classes. The main idea of the elaboration is to map a type class declaration onto a datatype that contains the method implementations, a so-called *(function) dictionary*.

```
> data EqD a = EqD { (==) :: a -> a -> Bool }
```

Then simple instances give rise to dictionary values:

```
> eqInt :: EqD Int
> eqInt = EqD { (==) = primEqInt }
```

Instances with a non-empty context are translated to functions that take context dictionaries to the instance dictionary.

```
> eqPair :: (EqD a, EqD b) -> Eq (a,b)
> eqPair (da, db) =
>   EqD { (==) = \(x1,y1) (x2,y2) -> (==) da x1 x2 && (==) db y1 y2 }
```

Functions with qualified types, like `refl`, are translated to functions that take explicit dictionaries as arguments.

```
> refl :: EqD a -> a -> Bool
> refl d x = (==) d x x
```

Finally, calls to functions with a qualified type are mapped to calls that explicitly pass the appropriate dictionary.

```
> main :: Bool
> main = refl (eqPair eqInt eqInt) (5,42)
```

Elaboration of Superclasses. Superclasses require a small extension to the above elaboration scheme. Consider the small program in Example 2 where `Sub1` is a subclass of `Base`. The function `test1` has `Sub1 a` in the context and calls `sub1` and `base` in its definition.

The standard approach to encode superclass is to embed the superclass dictionary in that of the subclass. For this case, `Sub1D a` contains a field `super1` that points to the superclass:

```
> class Base a where
>   base :: a -> Bool
>
> class Base a => Sub1 a where
>   sub1 :: a -> Bool
>
> test1 :: Sub1 a => a -> Bool
> test1 x = sub1 x && base x
```

Example 2. Program with superclasses.

```
> data BaseD a = BaseD { base :: a -> Bool }
> data Sub1D a = Sub1D { super1 :: BaseD a
>                      , sub1 :: a -> Bool }
```

This way we can extract the superclass from the subclass when needed. The function `test1` is then encoded as:

```
> test1 :: Sub1 a -> a -> Bool
> test1 d x = sub1 d x && base (super1 d) x
```

Resolution. Calls to functions with a qualified type generate type class constraints. The process for checking whether these constraints can be satisfied, is known as *resolution*. For the sake of dictionary-passing elaboration, this resolution process is augmented with the construction of the appropriate dictionary that witnesses the satisfiability of the constraint.

2.2 Nondeterminism and Coherence

For Haskell'98 programs there is usually only one way to construct a dictionary for a type class constraint. Yet, in the presence of superclasses, there may be multiple ways. Suppose we extend Example 2 with an additional subclass and the following function:

```
> class Base a => Sub2 a where
>   sub2 :: a -> Bool
>
> test2 :: (Sub1 a, Sub2 a) => a -> Bool
> test2 x = base x
```

There are two possible ways to resolve the `Base a` constraint that arises from the call to `base` in function `test2`, resulting in the following two translations: we can either establish the desired constraint as the superclass of the given `Sub1 a` constraint or as the superclass of the given `Sub2 a` constraint.

```
> test2a, test2b :: (Sub1D a, Sub2D a) -> a -> Bool
> test2a (d1,d2) x = base (super1 d1) x
> test2b (d1,d2) x = base (super2 d2) x
```

Fortunately, this nondeterminism is harmless because the difference between the two elaborations cannot be observed. Indeed, for any given type `A`, Haskell'98 only allows a single instance `Base A`, and it does not matter whether we access its dictionary directly or through one of its subclass instances. More generally, this suggests that type class resolution in Haskell'98 is coherent.

If we relax the Haskell'98 non-overlap condition for locally given constraints and adopt flexible contexts (allowing for arbitrary types in class constraints, rather than simple type variables), another source of nondeterminism arises. Consider:

```
> isZero :: Eq Int => Int -> Bool
> isZero n = n == 0
```

There are two ways to resolve the wanted `Eq Int` constraint that arises from the use of `(==)`. Either we use the global `Eq Int` constraint (in `isZero1`), or we use the locally given `Eq Int` constraint, passed as argument `d` (in `isZero2`):

```
> isZero1, isZero2 :: EqD Int -> Int -> Bool
> isZero1 d n = (==) eqInt n 0
> isZero2 d n = (==) d     n 0
```

Haskell'98 does not allow the `Eq Int` constraint in `isZero`'s signature, which overlaps with the global `Eq Int` instance; it only allows constraints on type variables in function signatures. This prevents the above nondeterminism in the elaboration. Yet, the nondeterminism is, once more, harmless; there is no way that the supplied dictionary `d` can be anything other than the global instance's dictionary `eqInt`. Informally, resolution remains coherent in the presence of flexible contexts.

2.3 Contextual Difference

While it is easy to provide an informal argument for the coherence of type class resolution, formally establishing the property is much harder. The indirect, elaboration-based attribution of a dynamic semantics in particular is a complicating factor, since it requires us to reason about two languages simultaneously. Unfortunately, there is another factor that further complicates the proof: different elaborations of the same source program can actually be distinguished in the target language. Consider, for instance, the target program below:

```
> discern :: ((Sub1D (), Sub2D ()) -> () -> Bool) -> Bool
> discern f =
>   let b1 = BaseD { base = \() -> True }
>       b2 = BaseD { base = \() -> False }
>       d1 = Sub1D { super1 = b1 }
>       d2 = Sub2D { super2 = b2 }
>   in f (d1,d2) ()
```

We find that `discern test2a` evaluates to `True` and `discern test2b` evaluates to `False`. Hence, since `discern` can differentiate between them, `test2a` and `test2b` clearly do not have the same meaning in the target language.

The dictionaries for `Sub1 ()` and `Sub2 ()` have different implementations for their `Base ()` superclass. The source language would never allow this, but the target language has no notion of type classes and happily admits `discern`'s violation of source language rules.

The problem is that the target language is more expressive than the source language. While `test2a` and `test2b` cannot be distinguished in any program context that arises from the source language, we can write target programs like `discern` that are not the image of any source program and thus do not have to play by the source language rules.

2.4 Our Approach to Proving Coherence

To avoid the problem with contextual difference in the target language, we employ a novel two-step approach. We prove that any elaboration from a source language program into a dictionary-passing encoding in the target language, can be decomposed in two separate elaborations through an intermediate language. We thus obtain two simpler problems for proving coherence of type class resolution.

The source language, λ_{TC} (presented in blue), features full-fledged type class resolution, and simplifies term typing with a bidirectional type system (a technique popularized by Pierce and Turner [2000]) to not distract from the main objective of coherent resolution.

The intermediate language, F_D (presented in green), is an extension of System F that explicitly passes type class dictionaries, and preserves the source language invariant that there is at most one such dictionary value for any combination of class and type. We show F_D is type-safe and strongly normalizing, and define a logical relation that captures the contextual equivalence of two F_D terms.

The target language, $F_{\{\}}$ (presented in red), is a different variant of System F without direct support for type class dictionaries; instead it features records, which can be used to encode dictionaries, but does not enforce uniqueness of instances.

The different calculi are presented in Figure 1, where the edges denote possible elaborations.

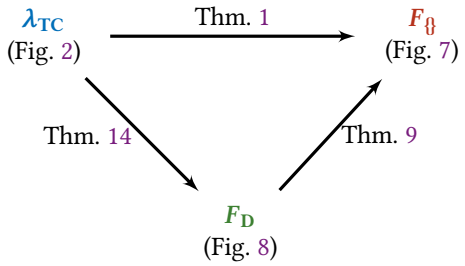

Fig. 1. The different calculi with elaborations

The coherence proof consists of two main parts:

Coherent Elaboration from λ_{TC} to F_D. Our elaboration from λ_{TC} into F_D is nondeterministic, but type preserving. Furthermore, we show that any two F_D elaborations of the same λ_{TC} term are logically related, and prove that this logical relation implies contextual equivalence. This establishes that the elaboration from λ_{TC} to F_D is coherent.

Deterministic Elaboration from F_D to $F_{\{\}}$. Because of the syntactic similarity between F_D and $F_{\{\}}$, the elaboration from the former into the latter is a more straightforward affair. In addition to being type preserving, it is also deterministic, and preserves contextual equivalence.

These results are easily combined to show the coherence of the elaboration from λ_{TC} to $F_{\{\}}$, which implies coherence of elaboration-based type class resolution. The full proofs can be found in the appendix. Note that the proofs depend on a number of standard boilerplate conjectures (e.g., substitution lemmas), which can be found in Sections 10.1 and 11.1 of the appendix.

3 SOURCE LANGUAGE λ_{TC}

This section presents our source language λ_{TC}, a basic calculus which isolates nondeterministic resolution. The calculus only supports features that are essential for type class resolution and its coherence.

Consequently, the language is strongly normalizing, and thus does not support recursive let expressions, mutual recursion or recursive methods. This is a sensible choice, as recursion does not affect the fundamentals of the coherence proof. This work could include recursion through step indexing [Ahmed 2006], a well-known technique, but this would significantly clutter the proof. Recursion is discussed in more detail in Section 8.

$$
\begin{array}{lll}
\tau & ::= & \mathit{Bool} \mid a \mid \tau_1 \to \tau_2 \qquad \textit{monotype} \\
\rho & ::= & \tau \mid Q \Rightarrow \rho \qquad \textit{qualified type} \\
\sigma & ::= & \rho \mid \forall a.\sigma \qquad \textit{type scheme} \\
\\
Q & ::= & TC\,\tau \qquad \textit{class constraint} \\
C & ::= & \forall \overline{a}.\overline{Q} \Rightarrow Q \qquad \textit{constraint scheme} \\
\\
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid m \mid \lambda x.e \mid e_1\, e_2 \mid \textbf{let}\ x : \sigma = e_1\ \textbf{in}\ e_2 \mid e :: \tau \qquad \textit{term} \\
\\
\mathit{pgm} & ::= & e \mid \mathit{cls}; \mathit{pgm} \mid \mathit{inst}; \mathit{pgm} \qquad \boldsymbol{\lambda_{TC}}\ \textit{program} \\
\mathit{cls} & ::= & \textbf{class}\ \overline{TC_i\ a} \Rightarrow TC\,a\ \textbf{where}\ \{m : \sigma\} \qquad \textit{class decl.} \\
\mathit{inst} & ::= & \textbf{instance}\ \overline{Q} \Rightarrow TC\,\tau\ \textbf{where}\ \{m = e\} \qquad \textit{instance decl.} \\
\\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q \qquad \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : \overline{TC_i\ a} \Rightarrow TC\,a : \sigma \qquad \textit{class environment} \\
P & ::= & \bullet \mid P, (D : C).m \mapsto \Gamma : e \qquad \textit{program context} \\
M & ::= & [\,\bullet\,] \mid \lambda x.M \mid M\,e \mid e\,M \mid M :: \tau \qquad \textit{evaluation context} \\
 & & \mid \textbf{let}\ x : \sigma = M\ \textbf{in}\ e \mid \textbf{let}\ x : \sigma = e\ \textbf{in}\ M
\end{array}
$$

Fig. 2. $\boldsymbol{\lambda_{TC}}$ syntax

Furthermore, two notable design decisions were made in the support of superclasses in $\boldsymbol{\lambda_{TC}}$. Firstly, similar to GHC, $\boldsymbol{\lambda_{TC}}$ derives all possible superclass constraints from their subclass constraints in advance, instead of deriving them "just-in-time" during resolution. The advantage of this approach is that it streamlines the actual resolution process.

Secondly, similar to Coq [Sozeau and Oury 2008] and unlike Wadler and Blott [1989], we pass superclass dictionaries alongside their subclass dictionaries, i.e., in a *flattened* form, instead of nesting them inside their subclass dictionaries. As it is not too difficult to see that both approaches are isomorphic, flattening the superclasses does not impact the coherence of resolution. It does, however, considerably simplify the proof, since this way neither our type class resolution mechanism, nor the intermediate language $\boldsymbol{F_D}$ (Section 6) need to have any support for superclasses and can treat them as regular local constraints. A more structured representation would give rise to additional complexity, but would not alter the essence of the proof.

Syntax. Figure 2 presents the, mostly standard, syntax. Programs consist of a number of class (with superclasses) and instance declarations, and an expression. For the sake of simplicity and well-foundedness, the declarations are ordered and can only refer to previous declarations.

Following Jones [1994]'s qualified types framework, we distinguish between three sorts of types: monotypes τ, qualified types ρ which include constraints, and type schemes σ which include type abstractions. Constraints differentiate between full constraint schemes C and simple class constraints Q. Observe that we allow flexible contexts in the qualified types; they are not restricted to constraints on type variables.

The definition of expressions e is standard, but with a few notable exceptions. Firstly, the language differentiates syntactically between regular variables x and method names m, which are introduced in class declarations. Secondly, type annotations $e :: \tau$ allow the programmer to manually assign a monotype to an expression. This is useful for resolving ambiguity—see the *Typing* paragraph below. Finally, let bindings include type annotations with a type scheme σ, allowing the programmer

to introduce local constraints—also discussed in the *Typing* paragraph. Note that we use Haskell syntax for class and instance declarations.

There are three $\boldsymbol{\lambda_{\mathbf{TC}}}$ environments: two global ones and one local environment. Firstly, the global class environment Γ_C stores all class declarations. Each entry in Γ_C contains the method name m, any superclasses $\overline{TC_i\ a}$, the class $TC\ a$ itself and the corresponding method type σ.

Secondly, the global program context P contains all instance declarations. Each entry in P consists of a unique dictionary constructor D, its corresponding constraint C, the method name m and its implementation e, together with the context Γ under which e should be interpreted. This context contains the local axioms available in this instance declaration, as well as any axioms which explicitly annotate the method type signature.

Thirdly, the local typing environment Γ, besides containing the default term and type variables x and a, also stores any local axioms Q. As opposed to the program context P, Γ does not contain any type class instances. Instead, the (local) axioms are associated with a dictionary variable δ. Sections 6 and 6.1 explain the use of these dictionaries.

Typing. Our type system features two design choices to eliminate the possibility of ambiguous type schemes. This allows us to focus on the coherence of type class resolution, by making our proof orthogonal to ambiguous type schemes, the source of ambiguity which has already been studied by Jones [1993]. We thus side-step an already solved problem and focus on tackling the full problem of resolution coherence.

Firstly, we require type signatures to be unambiguous (Figure 4, right-hand side) to make sure that all newly introduced type variables are bound in the head of the type (the remaining monotype after dropping all type and constraint abstractions). This prevents ambiguous expressions such as:

```
> let f : forall a . Eq a => Int -> Int -- ambiguous
>       = \ x . x + 1 in f 42
```

Secondly, we use a bidirectional type system rather than a fully declarative one. A bidirectional type system distinguishes between two typing modes: *inference* and *check* mode. The former synthesizes a type from the given expression, while the latter checks whether a given expression is of a given type. Special in our setting is that variables can only be typed in check mode, to ensure that only a single instantiation exists. This avoids the ambiguity that can arise when instantiating type variables in inference mode. Consider the following example:

```
> let y : forall a . Eq a => a -> a = ...
> in const 1 y
```

where `const x` is the constant function, which evaluates to `x` for any input. The instantiation of `y`'s type scheme is not uniquely determined by the context in which it is used. In a declarative type system or in inference mode, this ambiguity would result in multiple distinct typings and corresponding elaborations. While this ambiguity is harmless, it is not the focus of this work. Hence, to focus exclusively on the resolution, we use a bidirectional type system with check mode for variables to eliminate this irrelevant source of ambiguity.

Figure 3 [1] shows selected typing rules. The full set of rules can be found in Section 2.2 of the appendix. We ignore the red (elaboration-related) parts for now and explain them in detail in Section 4.1. The judgments $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau$ and $P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau$ denote *infer*ring a monotype τ for expression e and *check*ing e to have a monotype τ respectively, in environments P, Γ_C and Γ. Note that the constraint and type well-formedness relations $\vdash_Q$ and $\vdash_{ty}$ are omitted, as they are standard well-scopedness checks. They can be found in Section 2.1 of the appendix.

[1] Note that lists, such as $\overline{\tau}_i$, are denoted by overlines, whereas collections of predicates are annotated by their range. For instance, $(\Gamma_C; \Gamma \vdash_{ty} \tau_i \rightsquigarrow \sigma_i\ ,\ \forall i)$ iterates over i.

$\boxed{P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e}$ *(λ_{TC} Term Inference)*

$$\frac{\begin{array}{c} x \notin \mathbf{dom}(\Gamma) \qquad \mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1) \qquad \mathbf{closure}(\Gamma_C; \overline{Q}_i) = \overline{Q}_k \\ (\Gamma_C; \Gamma \vdash_Q Q_k \rightsquigarrow \sigma_k\ ,\ \forall k) \qquad \Gamma_C; \Gamma \vdash_{ty} \forall \overline{a}_j.\overline{Q}_k \Rightarrow \tau_1 \rightsquigarrow \forall \overline{a}_j.\overline{\sigma}_k \to \sigma \\ \overline{\delta}_k \ \mathbf{fresh} \qquad P; \Gamma_C; \Gamma, \overline{a}_j, \overline{\delta}_k : \overline{Q}_k \vdash_{tm} e_1 \Leftarrow \tau_1 \rightsquigarrow e_1 \\ P; \Gamma_C; \Gamma, x : \forall \overline{a}_j.\overline{Q}_k \Rightarrow \tau_1 \vdash_{tm} e_2 \Rightarrow \tau_2 \rightsquigarrow e_2 \end{array}}{\begin{array}{l} P; \Gamma_C; \Gamma \vdash_{tm} \mathbf{let}\ x : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau_1 = e_1\ \mathbf{in}\ e_2 \Rightarrow \tau_2 \\ \rightsquigarrow \mathbf{let}\ x : \forall \overline{a}_j.\overline{\sigma}_k \to \sigma_1 = \Lambda \overline{a}_j.\lambda\, \overline{\delta_k : \sigma_k}^{\,k}\ .e_1\ \mathbf{in}\ e_2 \end{array}}\ \textsc{sTm-infT-let}$$

$\boxed{P; \Gamma_C; \Gamma \vdash_{tm} e \Leftarrow \tau \rightsquigarrow e}$ *(λ_{TC} Term Checking)*

$$\frac{\begin{array}{c} (m : \overline{Q}'_k \Rightarrow TC\ a : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau') \in \Gamma_C \\ \mathbf{unambig}(\forall \overline{a}_j, a.\overline{Q}_i \Rightarrow \tau') \qquad P; \Gamma_C; \Gamma \vDash TC\ \tau \rightsquigarrow e \\ \Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma \qquad (P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j/\overline{a}_j][\tau/a]Q_i \rightsquigarrow e_i\ ,\ \forall i) \\ (\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j) \qquad \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \end{array}}{P; \Gamma_C; \Gamma \vdash_{tm} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow e.m\,\overline{\sigma}_j\,\overline{e}_i}\ \textsc{sTm-checkT-meth}$$

Fig. 3. λ_{TC} typing, selected rules

$\boxed{\mathbf{closure}(\Gamma_C; \overline{Q}_i) = \overline{Q}_j}$ *(Superclass Closure)*

$$\frac{}{\mathbf{closure}(\Gamma_C; \bullet) = \bullet}\ \textsc{sClosure-empty}$$

$$\frac{(m : \overline{Q}_m \Rightarrow TC\ a : \sigma) \in \Gamma_C \qquad \mathbf{closure}(\Gamma_C; \overline{Q}_i, \overline{Q}_m) = \overline{Q}_j}{\mathbf{closure}(\Gamma_C; \overline{Q}_i, TC\ a) = \overline{Q}_j, TC\ a}\ \textsc{sClosure-TC}$$

$\boxed{\mathbf{unambig}(\sigma)}$ *(Unambiguity for Type Schemes)*

$$\frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau)}\ \textsc{sUnambig-scheme}$$

$\boxed{\mathbf{unambig}(C)}$ *(Unambiguity for Constraints)*

$$\frac{\overline{a}_j \in \mathbf{fv}(\tau)}{\mathbf{unambig}(\forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\ \tau)}\ \textsc{sUnambig-constraint}$$

Fig. 4. Closure and unambiguity relations

$\boxed{P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow e}$ *(Constraint Entailment)*

$$\frac{(\delta : Q) \in \Gamma \qquad \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma}{P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow \delta}\ \textsc{sEntailT-local}$$

$$\frac{\begin{array}{c} P = P_1, (D : \forall \overline{a}_j.\overline{Q}'_i \Rightarrow Q').m \mapsto \Gamma' : e, P_2 \\ \Gamma' = \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}'_i, \overline{b}_k, \overline{\delta}_h : \overline{Q}_h \qquad Q = [\overline{\tau}_j/\overline{a}_j]Q' \qquad P_1; \Gamma_C; \Gamma' \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e \\ \vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma \qquad (\Gamma_C; \Gamma \vdash_{ty} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j) \qquad (\Gamma_C; \bullet, \overline{a}_j \vdash_Q Q'_i \rightsquigarrow \sigma'_i\ ,\ \forall i) \\ (\Gamma_C; \bullet, \overline{a}_j, \overline{b}_k \vdash_Q Q_h \rightsquigarrow \sigma''_h\ ,\ \forall h) \qquad (P; \Gamma_C; \Gamma \vDash [\overline{\tau}_j/\overline{a}_j]Q'_i \rightsquigarrow e_i\ ,\ \forall i) \end{array}}{P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow (\Lambda \overline{a}_j.\lambda\, \overline{\delta'_i : \sigma'_i}^{\,i}\ .\{m = \Lambda \overline{b}_k.\lambda\, \overline{\delta_h : \sigma''_h}^{\,h}\ .e\})\,\overline{\sigma}_j\,\overline{e}_i}\ \textsc{sEntailT-inst}$$

Fig. 5. λ_{TC} constraint entailment

$\boxed{P; \Gamma_C \vdash_{inst} inst : P'}$ *(Instance Decl Typing)*

$$\frac{\begin{array}{c}(m : \overline{Q}_i \Rightarrow TC\ a : \forall \overline{a}_j.\overline{Q}'_h \Rightarrow \tau_1) \in \Gamma_C \\ \overline{b}_k = \mathbf{fv}(\tau) \qquad \Gamma_C; \bullet, \overline{b}_k \vdash_{ty} \tau \rightsquigarrow \sigma \qquad \mathbf{closure}(\Gamma_C; \overline{Q}_p) = \overline{Q}_q \\ \mathbf{unambig}(\forall \overline{b}_k.\overline{Q}_q \Rightarrow TC\ \tau) \qquad (\Gamma_C; \bullet, \overline{b}_k \vdash_Q Q_q \rightsquigarrow \sigma_q\ ,\ \forall q) \\ D\ \mathbf{fresh} \qquad \overline{\delta}_q\ \mathbf{fresh} \qquad \overline{\delta}'_h\ \mathbf{fresh} \qquad (P; \Gamma_C; \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q \vDash [\tau/a]Q'_i \rightsquigarrow e_i\ ,\ \forall i) \\ \Gamma' = \bullet, \overline{b}_k, \overline{\delta}_q : \overline{Q}_q, \overline{a}_j, \overline{\delta}'_h : [\tau/a]\overline{Q}'_h \qquad P; \Gamma_C; \Gamma' \vdash_{tm} e \Leftarrow [\tau/a]\tau_1 \rightsquigarrow e \\ (D' : \forall \overline{b}'_m.\overline{Q}'_n \Rightarrow TC\ \tau_2).m' \mapsto \Gamma'' : e' \notin P \qquad \textit{where}\ [\overline{\tau}'_m/\overline{b}'_m]\tau_2 = [\overline{\tau}'_k/\overline{b}_k]\tau\end{array}}{P; \Gamma_C \vdash_{inst} \mathbf{instance}\ \overline{Q}_p \Rightarrow TC\ \tau\ \mathbf{where}\ \{m = e\} : (D : \forall \overline{b}_k.\overline{Q}_q \Rightarrow TC\ \tau).m \mapsto \Gamma' : e}\ \textsc{sInstT-inst}$$

Fig. 6. $\boldsymbol{\lambda_{TC}}$ instance declaration typing

Through a let binding (rule sTm-infT-let), the programmer provides a type scheme for a variable, thus potentially introducing local constraints. As explained above, the unambiguity check from Figure 4 (right-hand side) requires the provided type scheme to be unambiguous. In order to flatten the superclasses, the rule takes the closure over the superclass relation (left-hand side of Figure 4) of the user provided constraints $\overline{Q}_i$. It then adds the resulting set of constraints $\overline{Q}_k$ to the typing environment, under which to typecheck e_1. Finally, the type of e_2 is inferred under the extended environment.

Rule sTm-checkT-meth types a method call m in check mode, like regular variables, to avoid any ambiguity in the instantiation of the type variables in the method's type scheme. This includes both the type variable a from the class and any additional free variables $\overline{a}_j$ in the method type. Furthermore, the rules uses the unambig-relation to avoid ambiguity in the method type scheme itself, by requiring that both sets of type variables have to occur in the head of the method type. The rule also checks that all required constraints $\overline{Q}_i$ from the method type can be entailed.

The instance typing rule can be found in Figure 6. The relation $P; \Gamma_C \vdash_{inst} inst : P'$ denotes that an instance declaration $inst$ results in a $\boldsymbol{\lambda_{TC}}$ program context P', while being typed under environments P and Γ_C. The unambig-relation for constraints (Figure 4, bottom right), similarly to the unambig-relation for types, checks that all free type variables $\overline{b}_k$ in the instance context occur in the instance type τ as well, in order to avoid ambiguity. Like in the sTm-infT-let rule explained above, the superclasses of the instance context $\overline{Q}_p$ are flattened into additional local constraints $\overline{Q}_q$ and added to the environment. The superclasses $\overline{Q}'_i$ of the instantiated type class are then checked to be entailed under this extended environment. The rule checks that no overlapping instance declarations D' have been defined. Finally, the program context is extended with the new instance axiom D, consisting of a constraint scheme that requires the full set of local constraints $\overline{Q}_q$.

Type Class Resolution. The type class resolution rules can be found in Figure 5, where $P; \Gamma_C; \Gamma \vDash Q$ denotes that a class constraint Q is entailed under the environments P, Γ_C and Γ. A wanted constraint Q can either be resolved using a locally available constraint δ (sEntailT-local) or through a global instance declaration D (sEntailT-inst). The former is entirely straightforward. The latter is more involved as an instance D may have an instance context $\overline{Q}'_i$, which has to be recursively resolved. Before resolving the context, the type variables $\overline{a}_j$ are instantiated with the corresponding concrete types $\overline{\tau}_j$, originating from the wanted constraint Q.

Note that the type class resolution mechanism does not require any specific support for superclasses, as these have all been flattened into regular local constraints.

$$
\begin{array}{lll}
\sigma & ::= & \mathit{Bool} \mid a \mid \forall a.\sigma \mid \sigma_1 \to \sigma_2 \mid \{\, \overline{m_i : \sigma_i}^{\,i<n} \,\} & F_{\{\}}\ \mathit{type} \\
e & ::= & \mathit{True} \mid \mathit{False} \mid x \mid \lambda x : \sigma.e \mid e_1\, e_2 \mid \Lambda a.e \mid e\, \sigma & F_{\{\}}\ \mathit{term} \\
 & & \mid \{\, \overline{m_i = e_i}^{\,i<n} \,\} \mid e.m \mid \textbf{let}\ x : \sigma = e_1\ \textbf{in}\ e_2 & \\
\Gamma & ::= & \bullet \mid \Gamma, a \mid \Gamma, x : \sigma & F_{\{\}}\ \mathit{context}
\end{array}
$$

Fig. 7. Target language syntax

4 TARGET LANGUAGE $F_{\{\}}$

This section covers our target language $F_{\{\}}$, and the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$.

The target language is System F with records, which we consider a reasonable subcalculus of those used by Haskell compilers. Its syntax is shown in Figure 7. We omit its standard typing rules and call-by-name operational semantics and refer the reader to Pierce [2002, Chapter 23], or Section 5 of the appendix.

4.1 Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$

The red aspects in Figure 3 denote the elaboration of $\lambda_{\mathbf{TC}}$ terms to $F_{\{\}}$. We have adopted the convention that any red $F_{\{\}}$ types are the elaborated forms of their identically named blue $\lambda_{\mathbf{TC}}$ counterparts. This elaboration maps most $\lambda_{\mathbf{TC}}$ forms on identical $F_{\{\}}$ terms, with the exception of a few notable cases: (a) The interesting aspect of elaborating let expressions (sTm-infT-let) is that, as mentioned previously, superclasses are flattened into additional local constraints. The elaborated expression thus explicitly requires both the type variables and the full closure of the local constraints. (b) As opposed to $\lambda_{\mathbf{TC}}$, dictionary and type application are made explicit in $F_{\{\}}$. When elaborating variables x and method references m (sTm-checkT-meth), all previously substituted types $\overline{\tau}_j$ are now explicitly applied, together with the dictionary expressions $\overline{e}_i$. Furthermore, method names m are elaborated to $F_{\{\}}$ record labels m and therefore cannot appear by themselves, but must be applied to a record expression e, which originates from resolving the class constraint.

Type class resolution (Figure 5) of a $\lambda_{\mathbf{TC}}$ constraint Q results in a $F_{\{\}}$ expression e. When resolving the wanted constraint using a locally available constraint δ (sEntailT-local), this results in a regular term variable δ (which keeps the name of its $\lambda_{\mathbf{TC}}$ counterpart for readability). On the other hand, when resolving with the use of a global instance declaration D (sEntailT-inst), a record expression is constructed, containing the method name m and its corresponding implementation e. This method implementation now explicitly abstracts over the type variables $\overline{b}_k$ and term variables $\overline{\delta}_h$ originating from the method types's class constraints $\overline{Q}_h$, which annotate the class declaration. Furthermore, the record expression is nested in abstractions over the type variables $\overline{a}_j$ and term variables $\overline{\delta}'_i$ arising from the corresponding instance constraints $\overline{Q}'_i$. These abstractions are immediately instantiated by applying (a) the types $\overline{\sigma}_j$ needed for matching the wanted constraint Q to the instance declaration Q' and (b) the expressions $\overline{e}_i$ constructed by resolving the instance context constraints $\overline{Q}'_i$.

Example 1 $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$. Typing the Example 1 program results in the following environments:

$$\Gamma_C = (==) : \mathit{Eq}\, a : a \to a \to \mathit{Bool}$$

$$
\begin{aligned}
P = {} & (D_1 : \mathit{Eq}\, \mathit{Int}).(==) \mapsto \bullet : \mathit{primEqInt} \\
& , (D_2 : \forall a, b.\mathit{Eq}\, a \Rightarrow \mathit{Eq}\, b \Rightarrow \mathit{Eq}\, (a, b)).(==) \mapsto a, b, \delta_1 : \mathit{Eq}\, a, \delta_2 : \mathit{Eq}\, b : \\
& \quad \lambda(x_1, y_1).\lambda(x_2, y_2).\ (\&\&)\ ((==)\, x_1\, x_2)\ ((==)\, y_1\, y_2)
\end{aligned}
$$

The Eq class straightforwardly gets stored in the class environment Γ_C. Instances are stored in the $\boldsymbol{\lambda_{TC}}$ program context P (containing the dictionary constructor, the corresponding constraint, the method implementation and the environment under which to interpret this expression). Storing the instance declaration for $Eq\ Int$ is clear-cut. The instance for tuples on the other hand is somewhat more complex, since it requires an instance context, containing the local constraints $Eq\ a$ and $Eq\ b$. These constraints are made explicit, that is, the corresponding dictionaries are required by the elaborated implementation.

Elaborating the $\boldsymbol{\lambda_{TC}}$ expression results in the following $\boldsymbol{F_{\{\}}}$ expression:

$$
\begin{aligned}
&\textbf{let}\ \mathit{refl}\colon \forall a.\{(==) : a \to a \to \mathit{Bool}\} \to a \to \mathit{Bool}\\
&\qquad = \Lambda a.\lambda\delta_3 : \{(==) : a \to a \to \mathit{Bool}\}.\lambda x : a.\delta_3.(==)\,x\,x\\
&\textbf{in}\ \textbf{let}\ \mathit{main}\colon \mathit{Bool}\\
&\qquad\qquad = \mathit{refl}\,(\mathit{Int}, \mathit{Int})\\
&\qquad\qquad\quad (\Lambda a.\Lambda b.\lambda\delta_4 : \{(==) : a \to a \to \mathit{Bool}\}.\lambda\delta_5 : \{(==) : b \to b \to \mathit{Bool}\}.\\
&\qquad\qquad\qquad \{(==) = \lambda(x_1, y_1) : (a, b).\lambda(x_2, y_2) : (a, b).\\
&\qquad\qquad\qquad\quad (\&\&)\,(\delta_4.(==)\,x_1\,x_2)\,(\delta_5.(==)\,y_1\,y_2)\})\\
&\qquad\qquad\quad \mathit{Int}\ \mathit{Int}\ \{(==) = \mathit{primEqInt}\}\ \{(==) = \mathit{primEqInt}\}\ (5, 42)\\
&\textbf{in}\ \mathit{main}
\end{aligned}
$$

Note that the $Eq\ a$ constraint is made explicit in the implementation of *refl*, by abstracting over the constraint (elaborated to $\boldsymbol{F_{\{\}}}$ as the record type $\{(==) : a \to a \to \mathit{Bool}\}$, which stores the method name and its corresponding type) with the use of the record variable δ_3. When this function is called in *main*, both the type and the dictionary variable are instantiated. The latter is performed by (recursively) constructing a dictionary expression, using the type class resolution mechanism, as explained above in Section 4.1.

Example 2 $\boldsymbol{\lambda_{TC}}$ to $\boldsymbol{F_{\{\}}}$. Below is the environment generated by typing the Example 2 program (including the Section 2.2 extension), which features superclasses.

$$
\begin{aligned}
\Gamma_C = {} & \mathit{base} : \mathit{Base}\ a : a \to \mathit{Bool}\\
& , \mathit{sub}_1 : \mathit{Base}\ a \Rightarrow \mathit{Sub}_1\ a : a \to \mathit{Bool}\\
& , \mathit{sub}_2 : \mathit{Base}\ a \Rightarrow \mathit{Sub}_2\ a : a \to \mathit{Bool}
\end{aligned}
$$

The class environment Γ_C contains three classes, two of which have superclasses. However, since the example does not contain any instance declarations, the resulting program context P is empty.

For space reasons, we focus solely on elaborating `test2`, which results in the following $\boldsymbol{F_{\{\}}}$ expression:

$$
\begin{aligned}
&\textbf{let}\ \mathit{test}_2\colon \forall a.\{\mathit{base} : a \to \mathit{Bool}\} \to \{\mathit{sub}_1 : a \to \mathit{Bool}\}\\
&\qquad\qquad \to \{\mathit{base} : a \to \mathit{Bool}\} \to \{\mathit{sub}_2 : a \to \mathit{Bool}\} \to a \to \mathit{Bool}\\
&\qquad = \Lambda a.\lambda\delta_1 : \{\mathit{base} : a \to \mathit{Bool}\}.\lambda\delta_2 : \{\mathit{sub}_1 : a \to \mathit{Bool}\}.\\
&\qquad\qquad \lambda\delta_3 : \{\mathit{base} : a \to \mathit{Bool}\}.\lambda\delta_4 : \{\mathit{sub}_2 : a \to \mathit{Bool}\}.\lambda x : a.\ \delta_1.\mathit{base}\,x\\
&\textbf{in}\ \mathit{True}
\end{aligned}
$$

Note that the $\boldsymbol{\lambda_{TC}}$ expression requires two local constraints: $\mathit{Sub}_1\ a$ and $\mathit{Sub}_2\ a$. However, after flattening the superclasses and adding them to the local constraints, the elaborated $\boldsymbol{F_{\{\}}}$ expression requires (the elaborated form of) the $\mathit{Base}\ a$, $\mathit{Sub}_1\ a$, $\mathit{Base}\ a$ and $\mathit{Sub}_2\ a$ constraints. Notice the duplicate $\mathit{Base}\ a$ entry. Either of these two entries can be used for calling the method *base*. We have

arbitrarily selected the first here. The next section proves that both options are equivalent and can be used interchangeably.

5 COHERENCE

This section provides an outline for our coherence proof, and defines the required notions. We first provide a definition of *contextual equivalence* [Morris Jr 1969], which captures that two expressions have the same meaning.

5.1 Contextual Equivalence

In order to formally discuss the concept of contextual equivalence, we first define the notion of an *expression context*.

Expression Contexts. An expression context M is an expression with a single hole, for which another expression e can be filled in, denoted as $M[e]$. The syntax can be found in Figure 2.

The typing judgment for an expression context M is of the form $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$. This means that for any expression e such that $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e$, we have $P; \Gamma_C; \Gamma' \vdash_{tm} M[e] \Rightarrow \tau' \rightsquigarrow e'$. Following regular $\boldsymbol{\lambda_{TC}}$ term typing, context typing spans all combinations of type inference and checking mode: $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$, $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Rightarrow \tau') \rightsquigarrow M$ and $M : (P; \Gamma_C; \Gamma \Leftarrow \tau) \mapsto (P; \Gamma_C; \Gamma' \Leftarrow \tau') \rightsquigarrow M$.

For example, the simplest expression context is the empty context $[\bullet] : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \Gamma \Rightarrow \tau) \rightsquigarrow [\bullet]$.

Now we can formally define contextual equivalence. Note that the small step operational semantics can be found in Section 5.4 of the appendix. The environment and type well-formedness judgments can be found in Sections 2.4 and 2.1 of the appendix respectively.

Definition 1 (Kleene Equivalence).
Two $F_{\{\}}$ *expressions* e_1 *and* e_2 *are Kleene equivalent, written* $e_1 \simeq e_2$,
if there exists a value v *such that* $e_1 \longrightarrow^* v$, *and* $e_2 \longrightarrow^* v$.

Definition 2 (Contextual Equivalence).
Two expressions $\Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Gamma \vdash_{tm} e_2 : \sigma$,
where $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C; \Gamma \vdash_{ty} \tau \rightsquigarrow \sigma$,
are contextually equivalent, written $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$,
if forall $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_1$
and $M : (P; \Gamma_C; \Gamma \Rightarrow \tau) \mapsto (P; \Gamma_C; \bullet \Rightarrow \mathit{Bool}) \rightsquigarrow M_2$
implies $M_1[e_1] \simeq M_2[e_2]$.

The definition is adapted from Harper [2016, Chapter 46]. Intuitively, contextual equivalence means that two open expressions are observationally indistinguishable, when used in any program that instantiates the expressions' free variables.

5.2 Coherence

We can now make a first attempt to prove that different translations of the same source program are contextually equivalent. The program typing judgment can be found in Section 2.2 of the appendix.

Theorem 1 (Coherence).
If $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$ *and* $\bullet; \bullet \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$
then $\Gamma_{C1} = \Gamma_{C2}, P_1 = P_2$ *and* $P_1; \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$.

We first set out to prove the simpler variant, which only considers expressions [2].

[2]Theorem 2 also has a type checking mode counterpart, which has been omitted here for space reasons.

$$
\begin{array}{lll}
\sigma, \tau & ::= & \ldots \mid Q \Rightarrow \sigma \qquad \textit{type} \\
Q & ::= & TC\,\sigma \qquad \textit{class constraint} \\
C & ::= & \forall \overline{a}.\overline{Q} \Rightarrow Q \qquad \textit{constraint} \\
\\
d & ::= & \delta \mid D\,\overline{\sigma}\,\overline{d} \qquad \textit{dictionary} \\
dv & ::= & D\,\overline{\sigma}\,\overline{dv} \qquad \textit{dictionary value} \\
e & ::= & \ldots \mid \lambda \delta : Q.e \mid e\,d \mid d.m \qquad \textit{expression} \\
\\
\Gamma & ::= & \bullet \mid \Gamma, x : \sigma \mid \Gamma, a \mid \Gamma, \delta : Q \qquad \textit{typing environment} \\
\Gamma_C & ::= & \bullet \mid \Gamma_C, m : TC\,a : \sigma \qquad \textit{class environment} \\
\Sigma & ::= & \bullet \mid \Sigma, (D : C).m \mapsto e \qquad \textit{method environment}
\end{array}
$$

Fig. 8. $F_{\mathbf{D}}$, selected syntax

Theorem 2 (Expression Coherence).
If $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P; \Gamma_C; \Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
then $P; \Gamma_C; \Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

The main requirement which makes type class resolution coherent is that type class instances do not overlap. However, since $F_{\{\}}$ uses records to encode dictionaries, the $F_{\{\}}$ language does not enforce this crucial uniqueness property. In order to prove Theorem 1, we introduce an additional intermediate language $F_{\mathbf{D}}$, which captures the invariant that type class instances do not overlap, and makes it explicit.

6 INTERMEDIATE LANGUAGE $F_{\mathbf{D}}$

This section presents our intermediate language $F_{\mathbf{D}}$. The language is modeled with three main goals in mind: (a) $F_{\mathbf{D}}$ should explicitly pass type class dictionaries, which are implicit in $\lambda_{\mathbf{TC}}$; (b) the $F_{\mathbf{D}}$ type system should capture the uniqueness of dictionaries, thus enforcing the elaboration from $\lambda_{\mathbf{TC}}$ to preserve full abstraction; and (c) $F_{\mathbf{D}}$ expressions should elaborate straightforwardly and deterministically to the target language $F_{\{\}}$ (System F with records, see Section 4).

To this end, $F_{\mathbf{D}}$ is an extension of System F, with built-in support for dictionaries. These dictionaries differ from those commonly used in Haskell compilers in that they are special constants rather than a record of method implementations. A separate global map Σ from dictionaries to method implementations gives access to the latter. Note that this setup does not allow programs to introduce new (and possibly overlapping) dictionaries dynamically. All dictionaries have to be provided upfront, where uniqueness is easily enforced.

Syntax. Figure 8 shows selected syntax of $F_{\mathbf{D}}$; the basic System F constructs are omitted and can be found in Section 1.2 of the appendix.

$F_{\mathbf{D}}$ introduces a new syntactic sort of dictionaries d that can either be a dictionary variable δ or a dictionary constructor D. A dictionary constructor has a number (possibly zero) of type and dictionary parameters and always appears in fully-applied form. Each constructor corresponds to a unique instance declaration, and is mapped to its method implementation by the global environment Σ.

Expressions have explicit application and abstraction forms for dictionaries. Furthermore, similarly to $F_{\{\}}$, method names can no longer be used on their own. Instead, they have to be applied explicitly to a dictionary, in the form $d.m$.

$F_{\mathbf{D}}$ types σ or τ are identical to the well-known System F types, with the addition of a special function type $Q \Rightarrow \sigma$ for dictionary abstractions.

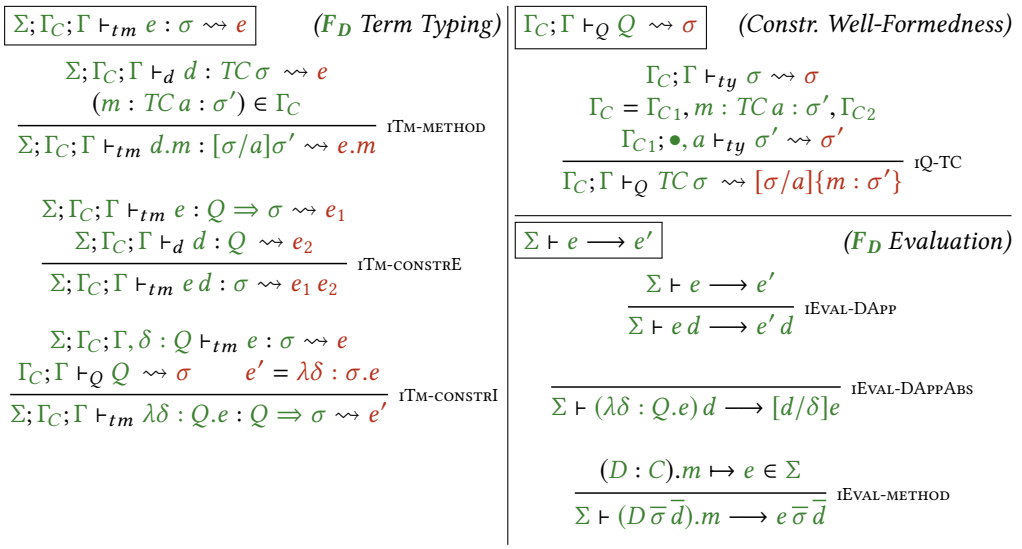

Fig. 9. F_D typing and operational semantics, selected rules

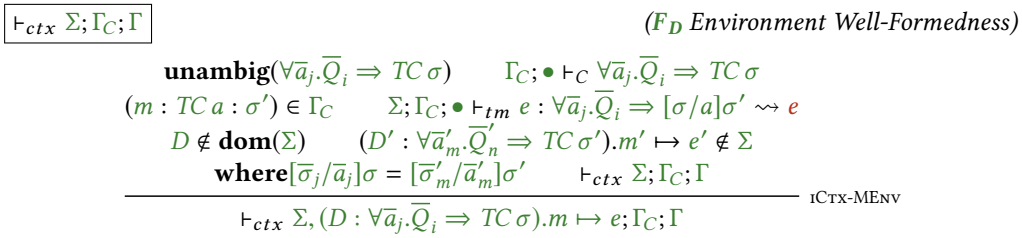

Fig. 10. F_D environment well-formedness, selected rules

Similarly to λ_{TC}, F_D features two global and a single local environment Γ. The latter is similar to the λ_{TC} typing environment Γ. However, there are notable differences between the global environments. The F_D class environment Γ_C does not contain any superclass information. The reason for this is that, as previously mentioned in Section 3, superclass constraints in the source language λ_{TC} are flattened into local constraints, and stored in the typing environment Γ. The analog to the λ_{TC} program context P is the F_D method environment Σ, storing information about all dictionary constructors D. Each constructor corresponds to a unique instance declaration, and stores the accompanying method implementations.

Typing. Figure 9 (left-hand side) shows selected typing rules for F_D expressions. The red parts can be safely ignored for now, as they will be explained in detail in Section 6.2. The judgment $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$ expresses that the F_D term e of type σ is well-typed under environments Σ, Γ_C and Γ. As shown by rule iTm-method, the type of a method variable applied to a dictionary is simply the corresponding method type (as stored in the static class environment), where the type variable has been substituted for the corresponding dictionary type.

Figure 11 shows the typing rules for dictionaries. The relation $\Sigma; \Gamma_C; \Gamma \vdash_d d : Q$ denotes that dictionary d of dictionary type Q is well-formed under environments Σ, Γ_C and Γ. Similarly to

$\boxed{\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e}$ *(Dictionary Typing)*

$$\frac{(\delta : Q) \in \Gamma \qquad \vdash_{ctx} \Sigma; \Gamma_C; \Gamma}{\Sigma; \Gamma_C; \Gamma \vdash_d \delta : Q \rightsquigarrow \delta} \text{ D-VAR}$$

$$\frac{\begin{array}{c}(D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{Q}_i.e \in \Sigma \\ (\Gamma_C; \bullet, \overline{a}_j \vdash_Q Q_i \rightsquigarrow \sigma'_i\ ,\ \forall i) \qquad (\Gamma_C; \Gamma \vdash_{ty} \sigma_j \rightsquigarrow \sigma_j\ ,\ \forall j) \\ \Sigma_1; \Gamma_C; \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i \vdash_{tm} e : [\sigma_q/a]\sigma_m \rightsquigarrow e \qquad (\Sigma; \Gamma_C; \Gamma \vdash_d d_i : [\overline{\sigma}_j/\overline{a}_j]Q_i \rightsquigarrow e_i\ ,\ \forall i) \\ \Sigma = \Sigma_1, (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow TC\,\sigma_q).m \mapsto \Lambda\overline{a}_j.\lambda\overline{\delta}_i : \overline{Q}_i.e, \Sigma_2\end{array}}{\Sigma; \Gamma_C; \Gamma \vdash_d D\,\overline{\sigma}_j\,\overline{d}_i : TC\,[\overline{\sigma}_j/\overline{a}_j]\sigma_q \rightsquigarrow (\Lambda\overline{a}_j.\lambda\,\overline{\delta_i : \sigma'_i}^{\,i}.\{m = e\})\,\overline{\sigma}_j\,\overline{e}_i} \text{ D-CON}$$

Fig. 11. F_D dictionary typing

regular term variables x (iTm-var), the type of a dictionary variable δ (D-var) is obtained from the typing environment Γ. The type of a dictionary constructor D (D-con), on the other hand, is obtained by finding the corresponding entry in the method environment Σ and substituting any types $\overline{\sigma}_j$ applied to it in the corresponding class constraint $TC\,\sigma_q$. All applied dictionaries $\overline{d}_i$ have to be well-typed with the corresponding constraint. Finally, the corresponding method implementation has to be well-typed in the reduced method environment Σ_1, which only contains the instances declared before D. As mentioned in Section 3, this reduced environment disallows recursive method implementations, as this would significantly clutter the coherence proof while, as a feature, recursion is completely orthogonal to the desired property.

Non-Overlapping Instances. The main requirement for achieving coherence of type class resolution, is that type class instances do not overlap. This requirement is common in Haskell and is for example enforced in GHC (though strongly discouraged, the `OverlappingInstances` pragma disables it). By storing all method implementations (with their corresponding instances) in a single environment Σ, this invariant can easily be made explicit.

Figure 10 shows the environment well-formedness condition $\vdash_{ctx} \Sigma; \Gamma_C; \Gamma$ for the method environment. Besides stating well-scopedness, it denotes that the method environment Σ cannot contain a second instance, for which the head of the constraint overlaps with $TC\,\sigma$, up to renaming. This key property will be exploited in our coherence proof.

Operational Semantics. As F_D is an extension of System F, its call-by-name operational semantics are mostly standard. The non-standard rules can be found in Figure 9 (bottom right), where $\Sigma \vdash e \longrightarrow e'$ denotes expression e evaluating to e' in a single step, under method environment Σ.

The evaluation rules for dictionary application (iEval-DApp and iEval-DAppAbs) are identical to those for term and type application. More interesting, however, is the evaluation for methods (iEval-method). A method name applied to a dictionary evaluates in one step to the method implementation, as stored in the environment Σ.

Metatheory. F_D is type safe. That is, the common progress and preservation properties hold:

Theorem 3 (Progress).
If $\Sigma; \bullet; \bullet \vdash_{tm} e : \sigma$, *then either* e *is a value, or there exists* e' *such that* $\Sigma \vdash e \longrightarrow e'$.

Theorem 4 (Preservation).
If $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$, *and* $\Sigma \vdash e \longrightarrow e'$, *then* $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e' : \sigma$.

$\boxed{P; \Gamma_C; \Gamma \vdash^M_{tm} e \Leftarrow \tau \rightsquigarrow e}$ *(Source Term Checking)*

$$\frac{\begin{array}{c}(m : \overline{Q}'_k \Rightarrow TC\ a : \forall \overline{a}_j.\overline{Q}_i \Rightarrow \tau') \in \Gamma_C \\ \textbf{unambig}(\forall \overline{a}_j, a.\overline{Q}_i \Rightarrow \tau') \qquad P; \Gamma_C; \Gamma \vDash^M TC\ \tau \rightsquigarrow d \\ \Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma \qquad (P; \Gamma_C; \Gamma \vDash^M [\overline{\tau}_j/\overline{a}_j][\tau/a]Q_i \rightsquigarrow d_i\ ,\ \forall i) \\ (\Gamma_C; \Gamma \vdash^M_{ty} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j) \qquad \vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma\end{array}}{P; \Gamma_C; \Gamma \vdash^M_{tm} m \Leftarrow [\overline{\tau}_j/\overline{a}_j][\tau/a]\tau' \rightsquigarrow d.m\,\overline{\tau}_j\,\overline{d}_i}\ \textsc{sTm-check-meth}$$

Fig. 12. $\lambda_{\mathbf{TC}}$ typing with elaboration to $F_{\mathbf{D}}$, selected rules

$\boxed{P; \Gamma_C; \Gamma \vDash^M Q \rightsquigarrow d}$ *(Constraint Entailment)*

$$\frac{(\delta : Q) \in \Gamma \qquad \vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma}{P; \Gamma_C; \Gamma \vDash^M Q \rightsquigarrow \delta}\ \textsc{sEntail-local}$$

$$\frac{\begin{array}{c}P = P_1, (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q').m \mapsto \Gamma' : e, P_2 \\ \Gamma' = \bullet, \overline{a}_j, \overline{\delta}_i : \overline{Q}_i, \overline{b}_k, \overline{\delta}_h : \overline{Q}_h \qquad Q = [\overline{\tau}_j/\overline{a}_j]Q' \qquad (\Gamma_C; \Gamma \vdash^M_{ty} \tau_j \rightsquigarrow \sigma_j\ ,\ \forall j) \\ \vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma \qquad (P; \Gamma_C; \Gamma \vDash^M [\overline{\tau}_j/\overline{a}_j]Q_i \rightsquigarrow d_i\ ,\ \forall i)\end{array}}{P; \Gamma_C; \Gamma \vDash^M Q \rightsquigarrow D\,\overline{\sigma}_j\,\overline{d}_i}\ \textsc{sEntail-inst}$$

Fig. 13. $\lambda_{\mathbf{TC}}$ constraint entailment with elaboration to $F_{\mathbf{D}}$

Analogously to $\lambda_{\mathbf{TC}}$, $F_{\mathbf{D}}$ rejects recursive expressions (including mutual recursion and recursive methods). This allows for a normalizing language, that is, any well-typed expression evaluates to a value, after a finite number of steps. Note that since the small step operational semantics are deterministic, normalization implies strong normalization.

Theorem 5 (Strong Normalization).
If $\Sigma; \Gamma_C; \bullet \vdash_{tm} e : \sigma$ *then all possible evaluation derivations for* e *terminate* : $\exists v : \Sigma \vdash e \longrightarrow^* v$.

The proof follows the familiar structure for proving normalization using logical relations, as presented by Ahmed [Ahmed 2015], and can be found in Section 11.4 of the appendix.

6.1 Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$

The green aspects in Figure 12 denote the elaboration of $\lambda_{\mathbf{TC}}$ terms to $F_{\mathbf{D}}$. Similarly to the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$, we have adopted the convention that any green $F_{\mathbf{D}}$ types or constraints are the elaborated forms of their identically named blue $\lambda_{\mathbf{TC}}$ counterparts. This elaboration works analogously to the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$, as shown in Figure 3. The full set of rules can be found in Section 3.2 of the appendix.

The only notable case is sTm-check-meth, where the entailment relation for solving the type class constraint $TC\ \tau$ now results in a dictionary d. As explained at the start of Section 6, unlike $F_{\{\}}$, $F_{\mathbf{D}}$ differentiates syntactically between dictionaries and normal expressions.

Type class resolution (Figure 13) of a $\lambda_{\mathbf{TC}}$ constraint Q results in a $F_{\mathbf{D}}$ dictionary d. When using a locally available constraint to resolve the wanted constraint (sEntail-local), the corresponding dictionary variable δ is returned. On the other hand, when resolving using a global instance declaration (sEntail-inst), a dictionary is constructed by taking the corresponding constructor D

and applying (a) the types $\overline{\sigma}_j$ needed for matching the wanted constraint to the instance declaration and (b) the dictionaries $\overline{d}_i$, constructed by resolving the instance context constraints.

Metatheory. We discuss the coherence of the elaboration from $\boldsymbol{\lambda_{TC}}$ to $\boldsymbol{F_D}$ in detail in Section 7, and mention here that it is type preserving:

Theorem 6 (Typing Preservation - Expressions).
If $P; \Gamma_C; \Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e$, *and* $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$, *and* $\Gamma_C; \Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$,
then $\Sigma; \Gamma_C; \Gamma \vdash_{tm} e : \sigma$.

The same theorem holds for check mode, but is omitted for space reasons. The full proofs can be found in Section 10.3 of the appendix.

Example 1 $\boldsymbol{\lambda_{TC}}$ *to* $\boldsymbol{F_D}$. Elaborating the $\boldsymbol{\lambda_{TC}}$ environments that originate from Example 1, results in the following $\boldsymbol{F_D}$ environments:

$$\Gamma_C = (==) : Eq\ a : a \to a \to Bool$$

$$\begin{aligned}\Sigma = {} & (D_1 : Eq\ Int).(==) \mapsto primEqInt \\ & , (D_2 : \forall a, b.Eq\ a \Rightarrow Eq\ b \Rightarrow Eq\ (a, b)).(==) \mapsto \\ & \quad \Lambda a.\Lambda b.\lambda \delta_1 : Eq\ a.\lambda \delta_2 : Eq\ b. \\ & \quad \lambda (x_1, y_1) : (a, b).\lambda (x_2, y_2) : (a, b).(\&\&)\ (\delta_1.(==)\ x_1\ x_2)\ (\delta_2.(==)\ y_1\ y_2)\end{aligned}$$

Note that both the class environment Γ_C and the program context Σ are largely direct translations of their $\boldsymbol{\lambda_{TC}}$ counterparts. One notable difference is the fact that the environment Γ under which to interpret the $\boldsymbol{\lambda_{TC}}$ method implementation is now explicitly abstracted over in the $\boldsymbol{F_D}$ method implementation. Consider for instance the case of D_2, where the variables a, b, δ_1 and δ_2, which in $\boldsymbol{\lambda_{TC}}$ are implicitly provided by the typing environment, are now explicit in the term level.

Elaborating the $\boldsymbol{\lambda_{TC}}$ expression results in the following $\boldsymbol{F_D}$ expression:

$$\begin{aligned}&\textbf{let}\ refl : \forall a.Eq\ a \Rightarrow a \to Bool \\ &\qquad = \Lambda a.\lambda \delta_3 : Eq\ a.\lambda x : a.\delta_3.(==)\ x\ x \\ &\textbf{in let}\ main : Bool \\ &\qquad = refl\ (Int, Int)\ (D_2\ Int\ Int\ D_1\ D_1)\ (5, 42) \\ &\textbf{in}\ main\end{aligned}$$

Unlike the corresponding $\boldsymbol{F_{\{\}}}$ expression, shown in Section 4.1, records storing the method types and implementations do not need to be passed around explicitly. In $\boldsymbol{F_D}$, they are replaced by class constraints and dictionaries, respectively. The construction of these dictionaries through type class resolution is shown in Figure 13.

Example 2 $\boldsymbol{\lambda_{TC}}$ *to* $\boldsymbol{F_D}$. Elaborating Example 2, including the extension from Section 2.2, results in the following $\boldsymbol{F_D}$ class environment (since no instance declarations exist, the method environment Σ remains empty):

$$\begin{aligned}\Gamma_C = {} & base : Base\ a : a \to Bool \\ & , sub_1 : Sub_1\ a : a \to Bool \\ & , sub_2 : Sub_2\ a : a \to Bool\end{aligned}$$

The $\boldsymbol{F_D}$ class environment no longer needs to store superclasses, as these are all flattened into additional local constraints during elaboration.

Similarly to Section 4.1, we focus solely on elaborating `test2`, which results in the following F_D expression:

$$\begin{aligned}&\textbf{let } test_2 \colon \forall a.Base\ a \Rightarrow Sub_1\ a \Rightarrow Base\ a \Rightarrow Sub_2\ a \Rightarrow a \to Bool\\ &\qquad = \Lambda a.\lambda\delta_1 : Base\ a.\lambda\delta_2 : Sub_1\ a.\lambda\delta_3 : Base\ a.\lambda\delta_4 : Sub_2\ a.\\ &\qquad\quad \lambda x : a.\delta_1.base\ x\\ &\textbf{in } True\end{aligned}$$

The only difference with the $F_{\{\}}$ elaboration is that we now use class constraints instead of passing around a record type (storing the method types).

6.2 Elaboration from F_D to $F_{\{\}}$

As both F_D and $F_{\{\}}$ are extensions of System F, the elaboration from former to latter is mostly trivial, leaving common features unchanged. The mapping of F_D dictionaries into $F_{\{\}}$ records, however, is non-trivial. Briefly, dictionary types are elaborated into record types, as shown in Figure 9 (top right), and dictionaries into record expressions, possibly nested within type and term abstractions and applications, as shown in Figure 11.

In particular, a dictionary type, TC, which corresponds to a unique entry $(m : TC\ a : \sigma')$ in the class environment Γ_C, elaborates to a record type whose field has the same name as the dictionary type's method name, m, and the type of that field is determined by the elaboration of σ'. A $TC\ \sigma$ dictionary elaborates to a record expression which is surrounded, firstly, by abstractions over type and term variables that arise from the method type's class constraints and, secondly, by type and term applications that properly instantiate those abstractions.

Metatheory. The following theorems confirm that the F_D-to-$F_{\{\}}$ elaboration is indeed appropriate.

The first theorem states that a well-typed F_D expression always elaborates to a $F_{\{\}}$ expression that is also well-typed in the translated context.

Theorem 7 (Type Preservation).
If $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$,
then there are unique Γ *and* σ *such that* $\Gamma \vdash_{tm} e : \sigma$,
where $\Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$ *and* $\Gamma_C;\Gamma \rightsquigarrow \Gamma$.

Secondly, and more importantly, the dynamic semantics is also preserved by the elaboration.

Theorem 8 (Semantic Preservation).
If $\Sigma;\Gamma_C;\bullet \vdash_{tm} e : \sigma \rightsquigarrow e$ *and* $\Sigma \vdash e \longrightarrow^* v$
then there exists a v *such that* $\Sigma;\Gamma_C;\bullet \vdash_{tm} v : \sigma \rightsquigarrow v$ *and* $e \simeq v$.

Thirdly, the elaboration is entirely deterministic.

Theorem 9 (Determinism).
If $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_1$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma \rightsquigarrow e_2$, *then* $e_1 = e_2$.

6.3 Elaboration Decomposition

An elaboration from λ_{TC} to $F_{\{\}}$ can always be decomposed into two elaborations through F_D. This intuition is formalized in Theorems 10 and 11 respectively.

Theorem 10 (Elaboration Equivalence - Expressions).
If $P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma$
then $P;\Gamma_C;\Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e$ *and* $\Sigma;\Gamma_C;\Gamma \vdash_{tm} e : \sigma \rightsquigarrow e$
where $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma$ *and* $\Gamma_C;\Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C;\Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$.

Theorem 11 (Elaboration Equivalence - Dictionaries).
If $P; \Gamma_C; \Gamma \vDash Q \rightsquigarrow e$ *and* $\vdash_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Gamma$
then $P; \Gamma_C; \Gamma \vDash^M Q \rightsquigarrow d$ *and* $\Sigma; \Gamma_C; \Gamma \vdash_d d : Q \rightsquigarrow e$
where $\vdash^M_{ctx} P; \Gamma_C; \Gamma \rightsquigarrow \Sigma; \Gamma_C; \Gamma$ *and* $\Gamma_C; \Gamma \rightsquigarrow \Gamma$.

Theorem 10 also has a type checking mode counterpart, which has been omitted for space reasons. The full proofs can be found in Section 12 of the appendix.

7 COHERENCE REVISITED

As mentioned previously in Section 5, the invariant that type class instances do not overlap is crucial in proving Theorem 1. This uniqueness property is made explicit in $F_{\mathbf{D}}$. Our proof thus proceeds by elaborating the $\lambda_{\mathbf{TC}}$ expression to two possibly different $F_{\mathbf{D}}$ expressions and subsequently elaborating these $F_{\mathbf{D}}$ expressions to $F_{\{\}}$ expressions. Consequently, the proof is split in two main steps. The first part is the most involved, where we use a technique based on logical relations to prove that any two $F_{\mathbf{D}}$ expressions originating from the same $\lambda_{\mathbf{TC}}$ expression are contextually equivalent. The second part proves that the elaboration from $F_{\mathbf{D}}$ to $F_{\{\}}$ is contextual equivalence preserving. This step follows straightforwardly from the fact that the $F_{\mathbf{D}}$-to-$F_{\{\}}$ elaboration is deterministic. Together, these prove that the elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\{\}}$ through $F_{\mathbf{D}}$ is coherent. Theorem 2 follows from this result, together with Theorem 10.

The remainder of this section explains the techniques we used to prove Theorem 1 in detail.

7.1 Coherent Elaboration from $\lambda_{\mathbf{TC}}$ to $F_{\mathbf{D}}$

7.1.1 Logical Relations. Logical relations [Plotkin 1973; Statman 1985; Tait 1967] are key to proving contextual equivalence. In our type system, the logical relation for expressions is mostly standard, though the relation for dictionaries is novel.

Dictionaries. The logical relation over two open dictionaries is defined by means of an auxiliary relation on closed dictionaries. We define this value relation for closed dictionaries as follows. Note that from now on, we will omit elaborations when they are entirely irrelevant. The appendix uses the same convention.

Definition 3 (Value Relation for Dictionaries).
The dictionary values $D\,\overline{\sigma}_j\,\overline{dv_{1\,i}}$ *and* $D\,\overline{\sigma}_j\,\overline{dv_{2\,i}}$ *are in the value relation, defined as:*

$$(\Sigma_1 : D\,\overline{\sigma}_j\,\overline{dv_{1\,i}}, \Sigma_2 : D\,\overline{\sigma}_j\,\overline{dv_{2\,i}}) \in \mathcal{V}[\![Q]\!]^{\Gamma_C}_R \triangleq ((\Sigma_1 : dv_{1\,i}, \Sigma_2 : dv_{2\,i}) \in \mathcal{V}[\![[\overline{\sigma}_j/\overline{a}_j]Q_i]\!]^{\Gamma_C}_R\ ,\ \forall i)$$
$$\wedge\ \ \Sigma_1; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_j\,\overline{dv_{1\,i}} : R(Q)\ \ \wedge\ \ \Sigma_2; \Gamma_C; \bullet \vdash_d D\,\overline{\sigma}_j\,\overline{dv_{2\,i}} : R(Q),$$
$$\textit{where}\ (D : \forall \overline{a}_j.\overline{Q}_i \Rightarrow Q').m \mapsto e_1 \in \Sigma_1 \wedge Q = [\overline{\sigma}_j/\overline{a}_j]Q'$$

The value relation is indexed by the dictionary type Q. We require both dictionaries to be well-typed, and their dictionary arguments to be in the value relation as well. The relation has four additional parameters: the contexts Σ_1 and Σ_2, which annotate the dictionaries, the class environment Γ_C, used in the well-typing condition, and the type substitution R.

In order to define logical equivalence between open dictionaries, we substitute all free variables with closed terms, thus reducing them to closed dictionary values. Three kinds of variables exist (term variables x, type variables a and dictionary variables δ). This results in three separate semantic interpretations of the typing context Γ. The type substitution $R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C}$ maps all type variables $a \in \Gamma$ onto closed types. $\phi \in \mathcal{G}[\![\Gamma]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_R$ maps each term variable $x \in \Gamma$ to two expressions e_1 and e_2 that are in the expression value relation (see Definition 5), and $\gamma \in \mathcal{H}[\![\Gamma]\!]^{\Sigma_1,\Sigma_2,\Gamma_C}_R$ maps each

dictionary variable $\delta \in \Gamma$ to two logically related dictionary values. We use ϕ_1 and ϕ_2 to denote the substitution for the first and second expression, respectively.

Definition 4 (Logical Equivalence for Dictionaries).
Two dictionaries d_1 *and* d_2 *are logically equivalent, defined as:*

$$\Gamma_C; \Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : Q \triangleq \forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C},\ \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C},\ \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} :$$
$$(\Sigma_1 : \gamma_1(\phi_1(R(d_1))), \Sigma_2 : \gamma_2(\phi_2(R(d_2)))) \in \mathcal{V}[\![Q]\!]_R^{\Gamma_C}$$

Two dictionaries d_1 and d_2 are logically equivalent if any substitution of their free variables (with related expressions / dictionaries) results in related dictionary values.

Expressions. The value relation for expressions is mostly standard, with two notable deviations. Firstly, the relation is defined over two different method environments Σ_1 and Σ_2. Hence, both expressions are annotated with their respective environment. Secondly, the dictionary abstraction case is novel.

Definition 5 (Value Relation for Expressions).
Two values v_1 *and* v_2 *are in the value relation, defined as:*

$$(\Sigma_1 : \mathit{True}, \Sigma_2 : \mathit{True}) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : \mathit{False}, \Sigma_2 : \mathit{False}) \in \mathcal{V}[\![\mathit{Bool}]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![a]\!]_R^{\Gamma_C} \triangleq$$
$$(a \mapsto (\sigma, r)) \in R \ \wedge\ (v_1, v_2) \in r \ \wedge\ \Sigma_1; \Gamma_C; \bullet \vdash_{tm} v_1 : \sigma \ \wedge\ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} v_2 : \sigma$$

$$(\Sigma_1 : \lambda x : \sigma_1.e_1, \Sigma_2 : \lambda x : \sigma_1.e_2) \in \mathcal{V}[\![\sigma_1 \to \sigma_2]\!]_R^{\Gamma_C} \triangleq$$
$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda x : \sigma.e_1 : R(\sigma_1 \to \sigma_2) \ \wedge\ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda x : \sigma.e_2 : R(\sigma_1 \to \sigma_2)$$
$$\wedge\ \forall (\Sigma_1 : e_3, \Sigma_2 : e_4) \in \mathcal{E}[\![\sigma_1]\!]_R^{\Gamma_C} : (\Sigma_1 : (\lambda x : \sigma.e_1)\, e_3, \Sigma_2 : (\lambda x : \sigma.e_2)\, e_4) \in \mathcal{E}[\![\sigma_2]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : \lambda \delta : Q.e_1, \Sigma_2 : \lambda \delta : Q.e_2) \in \mathcal{V}[\![Q \Rightarrow \sigma]\!]_R^{\Gamma_C} \triangleq$$
$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \lambda \delta : Q.e_1 : R(Q \Rightarrow \sigma) \ \wedge\ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \lambda \delta : Q.e_2 : R(Q \Rightarrow \sigma)$$
$$\wedge\ \forall (\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]_R^{\Gamma_C} : (\Sigma_1 : (\lambda \delta : Q.e_1)\, dv_1, \Sigma_2 : (\lambda \delta : Q.e_2)\, dv_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

$$(\Sigma_1 : \Lambda a.e_1, \Sigma_2 : \Lambda a.e_2) \in \mathcal{V}[\![\forall a.\sigma]\!]_R^{\Gamma_C} \triangleq$$
$$\Sigma_1; \Gamma_C; \bullet \vdash_{tm} \Lambda a.e_1 : R(\forall a.\sigma) \ \wedge\ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} \Lambda a.e_2 : R(\forall a.\sigma)$$
$$\wedge\ \forall \sigma', \forall r \in \mathit{Rel}[\sigma'] :\ \Gamma_C; \bullet \vdash_{ty} \sigma' \Rightarrow (\Sigma_1 : (\Lambda a.e_1)\, \sigma', \Sigma_2 : (\Lambda a.e_2)\, \sigma') \in \mathcal{E}[\![\sigma]\!]_{R, a \mapsto (\sigma', r)}^{\Gamma_C}$$

Consider the interesting case of dictionary abstraction. The relation requires the terms to be well-typed, and the applications for all related input dictionaries to be in the expression relation $\mathcal{E}$. The definition of this $\mathcal{E}$ relation is as follows:

Definition 6 (Expression Relation).
Two expressions e_1 *and* e_2 *are in the expression relation, defined as:*

$$(\Sigma_1 : e_1, \Sigma_2 : e_2) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C} \triangleq \Sigma_1; \Gamma_C; \bullet \vdash_{tm} e_1 : R(\sigma) \ \wedge\ \Sigma_2; \Gamma_C; \bullet \vdash_{tm} e_2 : R(\sigma)$$
$$\wedge\ \exists v_1, v_2, \Sigma_1 \vdash e_1 \longrightarrow^* v_1, \Sigma_2 \vdash e_2 \longrightarrow^* v_2, (\Sigma_1 : v_1, \Sigma_2 : v_2) \in \mathcal{V}[\![\sigma]\!]_R^{\Gamma_C}$$

In this definition, expressions are reduced to values and those values must be in the value relation. This is well-defined because F_D is strongly normalizing (Theorem 5).

Finally, we can give the definition of logical equivalence for open expressions:

Definition 7 (Logical Equivalence for Expressions).
Two expressions e_1 *and* e_2 *are logically equivalent, defined as:*

$$\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma \triangleq \forall R \in \mathcal{F}[\![\Gamma]\!]^{\Gamma_C},\ \phi \in \mathcal{G}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C},\ \gamma \in \mathcal{H}[\![\Gamma]\!]_R^{\Sigma_1,\Sigma_2,\Gamma_C} :$$
$$(\Sigma_1 : \gamma_1(\phi_1(R(e_1))), \Sigma_2 : \gamma_2(\phi_2(R(e_2)))) \in \mathcal{E}[\![\sigma]\!]_R^{\Gamma_C}$$

We also provide a definition of logical equivalence for contexts:

Definition 8 (Logical Equivalence for Contexts).
Two contexts M_1 *and* M_2 *are logically equivalent, defined as:*

$$\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Gamma_C;\Gamma' \Rightarrow \sigma') \triangleq$$
$$\forall e_1, e_2 : \Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma \implies \Gamma_C;\Gamma' \vdash \Sigma_1 : M_1[e_1] \simeq_{log} \Sigma_2 : M_2[e_2] : \sigma'$$

7.1.2 Proof of λ_{TC}-to-F_D Coherence. With the above definitions we are ready to formally state the metatheory and establish the coherence theorems from λ_{TC} to F_D.

Design Principle of F_D. We emphasize that F_D captures the intention of type class instances. Theorem 12 states that any two dictionary values for the same constraint are logically related:

Theorem 12 (Value Relation for Dictionary Values).
If $\Sigma_1;\Gamma_C;\bullet \vdash_d dv_1 : Q$ *and* $\Sigma_2;\Gamma_C;\bullet \vdash_d dv_2 : Q$ *and* $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$
then $(\Sigma_1 : dv_1, \Sigma_2 : dv_2) \in \mathcal{V}[\![Q]\!]_\bullet^{\Gamma_C}$.

Note that two environments Σ_1 and Σ_2 are logically equivalent under Γ_C, written $\Gamma_C \vdash \Sigma_1 \simeq_{log} \Sigma_2$, when they contain the same dictionary constructors and the corresponding method implementations are logically equivalent. The full definition can be found in Section 7.3 of the appendix.

Coherent Resolution. We now prove that constraint resolution is semantically coherent, that is, if multiple resolutions of the same constraint exist, they are logically equivalent.

Theorem 13 (Logical Coherence of Dictionary Resolution).
If $P;\Gamma_C;\Gamma \vDash^M Q \rightsquigarrow d_1$ *and* $P;\Gamma_C;\Gamma \vDash^M Q \rightsquigarrow d_2$
and $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_1;\Gamma_C;\Gamma$ *and* $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_2;\Gamma_C;\Gamma$
then $\Gamma_C;\Gamma \vdash \Sigma_1 : d_1 \simeq_{log} \Sigma_2 : d_2 : Q$ *where* $\Gamma_C;\Gamma \vdash^M_Q Q \rightsquigarrow Q$.

Coherent Elaboration. Furthermore, in order to prove that the elaboration from λ_{TC} to F_D is coherent, we show that all elaborations of the same expression are logically equivalent [3].

Theorem 14 (Logical Coherence of Expression Elaboration).
If $P;\Gamma_C;\Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P;\Gamma_C;\Gamma \vdash^M_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
and $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_1;\Gamma_C;\Gamma$ *and* $\vdash^M_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_2;\Gamma_C;\Gamma$
then $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *where* $\Gamma_C;\Gamma \vdash^M_{ty} \tau \rightsquigarrow \sigma$.

Contextual Equivalence. We prove that all logically equivalent expressions are contextually equivalent. Together with Theorem 14, this shows coherence of the λ_{TC}-to-F_D elaboration.

We first provide a formal definition of contextual equivalence for F_D expressions. Kleene equivalence for F_D is defined similarly to Definition 1 and can be found in Section 9.1 of the appendix.

[3] Theorem 14 also has a type checking mode counterpart, which has been omitted here for space reasons.

Definition 9 (Contextual Equivalence for F_D Expressions).
Two expressions $\Sigma_1;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Sigma_2;\Gamma_C;\Gamma \vdash_{tm} e_2 : \sigma$,
are contextually equivalent, written $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$,
if forall $M_1 : (\Sigma_1;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_1;\Gamma_C;\bullet \Rightarrow Bool)$
and forall $M_2 : (\Sigma_2;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_2;\Gamma_C;\bullet \Rightarrow Bool)$
where $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Gamma_C;\bullet \Rightarrow Bool)$,
we have that $\Sigma_1 : M_1[e_1] \simeq \Sigma_2 : M_2[e_2]$.

Theorem 15 (Logical Equivalence implies Contextual Equivalence).
If $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{log} \Sigma_2 : e_2 : \sigma$ *then* $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$.

7.2 Deterministic Elaboration from F_D to $F_{\{\}}$

7.2.1 Contextual Equivalence. Similarly to expressions, the elaboration from a λ_{TC} context M to a $F_{\{\}}$ context M can always be decomposed into two elaborations, through a F_D context M. The syntax and typing judgments can be found in Sections 1 and 6 of the appendix, respectively.

We now formally define contextual equivalence for $F_{\{\}}$ expressions through F_D contexts.

Definition 10 (Contextual Equivalence in F_D Context).
Two expressions $\Gamma \vdash_{tm} e_1 : \sigma$ *and* $\Gamma \vdash_{tm} e_2 : \sigma$,
where $\vdash_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C;\Gamma \vdash_{ty} \tau \rightsquigarrow \sigma$,
are contextually equivalent, written $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$,
if forall $M_1 : (\Sigma_1;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_1;\Gamma_C;\bullet \Rightarrow Bool) \rightsquigarrow M_1$
and forall $M_2 : (\Sigma_2;\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Sigma_2;\Gamma_C;\bullet \Rightarrow Bool) \rightsquigarrow M_2$
where $\Sigma_1 : M_1 \simeq_{log} \Sigma_2 : M_2 : (\Gamma_C;\Gamma \Rightarrow \sigma) \mapsto (\Gamma_C;\bullet \Rightarrow Bool)$,
we have that $M_1[e_1] \simeq M_2[e_2]$,
where $\vdash^{M}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma;\Gamma_C;\Gamma$ *and* $\Gamma_C;\Gamma \rightsquigarrow \Gamma$ *and* $\Gamma_C;\Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$ *and* $\Gamma_C;\Gamma \vdash_{ty} \sigma \rightsquigarrow \sigma$.

7.2.2 Proof of F_D-to-$F_{\{\}}$ Coherence. We continue by proving that contextual equivalence is preserved by the elaboration from F_D to $F_{\{\}}$:

Theorem 16 (Elaboration preserves Contextual Equivalence).
If $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$
and $\Sigma_1;\Gamma_C;\Gamma \vdash_{tm} e_1 : \sigma \rightsquigarrow e_1$ *and* $\Sigma_2;\Gamma_C;\Gamma \vdash_{tm} e_2 : \sigma \rightsquigarrow e_2$ *and* $\Gamma_C;\Gamma \rightsquigarrow \Gamma$
then $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$.

7.2.3 Proof of λ_{TC}-to-$F_{\{\}}$ Coherence. Finally, in order to link back to Theorem 2 (which has no notion of F_D), we prove that contextual equivalence with F_D contexts implies contextual equivalence with λ_{TC} contexts:

Theorem 17 (Contextual Equivalence in F_D implies Contextual Equivalence in λ_{TC}).
If $\Gamma_C;\Gamma \vdash \Sigma_1 : e_1 \simeq_{ctx} \Sigma_2 : e_2 : \sigma$
and $\vdash^{M}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_1;\Gamma_C;\Gamma$ *and* $\vdash^{M}_{ctx} P;\Gamma_C;\Gamma \rightsquigarrow \Sigma_2;\Gamma_C;\Gamma$ *and* $\Gamma_C;\Gamma \vdash^{M}_{ty} \tau \rightsquigarrow \sigma$
then $P;\Gamma_C;\Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

For clarity, we restate coherence Theorems 2 and 1:

Theorem 2 (Expression Coherence - Restated).
If $P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_1$ *and* $P;\Gamma_C;\Gamma \vdash_{tm} e \Rightarrow \tau \rightsquigarrow e_2$
then $P;\Gamma_C;\Gamma \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Theorem 2 follows by combining Theorems 10, 14, 15, 16 and 17.

Theorem 1 (Coherence - Restated).
If $\bullet;\bullet \vdash_{pgm} pgm : \tau; P_1; \Gamma_{C1} \rightsquigarrow e_1$ *and* $\bullet;\bullet \vdash_{pgm} pgm : \tau; P_2; \Gamma_{C2} \rightsquigarrow e_2$
then $\Gamma_{C1} = \Gamma_{C2}, P_1 = P_2$ *and* $P_1; \Gamma_{C1}; \bullet \vdash e_1 \simeq_{ctx} e_2 : \tau$.

Finally, we show that Theorem 1 follows from Theorem 2. The full proofs can be found in Section 13 of the appendix.

8 DISCUSSION OF POSSIBLE EXTENSIONS

As the goal of this work was to find a proof technique to formally establish coherence for type class resolution, a stripped down source calculus was employed in order not to clutter the proof. This section provides a brief discussion of extending our coherence proof to support several mainstream language features.

Ambiguous Type Schemes. As mentioned previously, our work is orthogonal to ambiguous type schemes, which have already been extensively studied by Jones [1993]. We believe our work and the proof by Jones can be combined, which would then relax the restriction of bidirectional type checking, and prove coherence for both ambiguous type schemes and type class resolution.

General Recursion. Recursion is an important feature, present in any real world programming language. It is important to note that, while λ_{TC} does not feature recursion on the expression level (as it does not affect the essence of the coherence proof), type class resolution itself is recursive. Dictionary values are constructed dynamically from a statically given set of dictionary constructors (one constructor per type class instance). The system can thus recursively generate an arbitrary number of dictionaries from a finite set of instances.

Our logical relations can be adapted to support general recursion, through well-known techniques, such as step indexing [Ahmed 2006]. While this results in a significantly longer and more cluttered proof, we do not anticipate any major complications.

Multi-Parameter Type Classes. Just like regular type class instances, multi-parameter instances (as supported by GHC) are subject to the no-overlap rule. Hence, they respect our main assumption. They may indeed give rise to more ambiguity, but this is the kind of ambiguity that is studied by Jones [1993], not the kind that shows up during resolution. Note that functional dependencies were originally introduced by Jones [2000] as a way to resolve the ambiguity caused by multi-parameter instances.

Dependent Types. Dependently typed languages, e.g., Agda [Devriese and Piessens 2011] and Idris [Brady 2013], include language features that are inspired by type classes. Proving resolution coherence in a dependently typed setting requires significant extension of our calculi, as dependent types collapse the term and type levels into a single level and thus enable more powerful type signatures for classes and instances. Furthermore, our logical relation needs to be extended to support dependent types [Bernardy et al. 2012] as well. Fortunately, the essence of our proof strategy still applies. That is, the intermediate language incorporates separate binding structures for dictionaries, and enforces the uniqueness of dictionaries. We thus believe a non-trivial extension of our proof methodology can be used to prove coherence for type class resolution in the setting of dependently typed languages.

Non-overlapping Instances. Our work is built on top of the assumption that type class instances do not overlap. This is enforced during the type checking of instance declarations, and made explicit in the intermediate language. Whether a constraint is entailed directly from an instance, through user provided constraints in a type annotation, or through local evidence, is not actually relevant, as all evidence ultimately has to originate from a non-overlapping instance declaration.

Therefore, our work can be extended to include features where the assumption holds true. This includes, among others, GADT's [Peyton Jones et al. 2006], implication constraints [Bottu et al. 2017], type constructors, higher kinded types and constraint kinds [Orchard and Schrijvers 2010], e.g., Bottu et al. [2017] informally discuss the coherence of implication constraints based on the same assumption. These features are all included in GHC.

Modules. Modules, as supported by GHC, pose an interesting challenge, as they are known to cause a form of ambiguity.[4] GHC does not statically check the uniqueness of instances across modules, thus indirectly allowing users to write overlapping instances, as long as no ambiguity arises during resolution. Adapting our global uniqueness assumption to accommodate this additional freedom remains an interesting challenge.

Laziness. The operational semantics of the $F_{\mathbf{D}}$ and $F_{\{\}}$ calculi in this work are given through standard call-by-name semantics, in order to approximate Haskell's laziness. The system can easily be adapted to either call-by-value or call-by-need, with little impact on the proofs.

It is important to note though, that while expressions are evaluated lazily, type class resolution itself is eager, and constructs the full dictionaries at compile time. This complicates supporting certain GHC features that rely on laziness, like cyclic and infinite dictionaries. They could be supported through loop detection and deferring the construction of dictionaries to runtime, but these would nonetheless pose a significant challenge.

9 RELATED WORK

Type Classes. Jones [1993, 1994] formally proves coherence for the framework of qualified types, which generalizes from type classes to arbitrary evidence-backed type constraints. He focuses on nondeterminism in the typing derivation, and assumes that resolution is coherent.

Morris [2014] presents an alternative, denotational semantics for type classes (without superclasses) that avoids elaboration and instead interprets qualified type schemes as the set of denotations of all its monomorphic instantiations that satisfy the qualifiers. The nondeterminism of resolution does not affect these semantics.

Kahl and Scheffczyk [2001] present named type class instances that are not used during resolution, but can be explicitly passed to functions. Nevertheless, they violate the uniqueness of instances, and give rise to incoherence of the form illustrated by our `discern` function in Section 2.3.

Unlike most other languages with type classes (such as Haskell, Mercury or PureScript) Coq [Sozeau and Oury 2008] does not enforce the non-overlapping instances condition. Consequently, coherence does not hold for type class resolution in Coq. The reason for this alternative design choice is twofold: (a) Since Coq's type system is more complex than that found in regular programming languages, it is not always possible to decide whether two instances overlap [Lampropoulos and Pierce 2018, Chapter 2: Typeclasses]. (b) Type class members in Coq are often proofs and, unlike for expressions, users are often indifferent to coherence in the presence of proofs (even though from a semantic point of view, Coq differentiates between them). This concept is known as "proof irrelevance" [Gilbert et al. 2019], that is, as long as at least one proof exists, the concrete choice between these proofs is irrelevant. Users can deal with this lack of coherence by either assigning priorities to overlapping instances, or by manually curating the instance database and locally removing specific instances.

Winant and Devriese [2018] introduce explicit dictionary application to the Haskell language, and prove coherence for this extended system. Their proof is parametric in the constraint entailment judgment and thus assumes that the constraint solver produces "canonical" evidence. They proceed

[4] http://blog.ezyang.com/2014/07/type-classes-confluence-coherence-global-uniqueness/

by introducing a disjointness condition to explicitly applied dictionaries, in order to ensure that coherence is preserved by their extension. Our paper proves their aforementioned assumption, by establishing coherence for type class resolution.

Dreyer et al. [2007] blend ML modules with Haskell type class resolution. Unlike Haskell, they feature multiple global (or outer) scopes; instances within one such global scope must not overlap. Moreover, global instances are shadowed by those given through type signatures. While their language has been formalized, no formal proof of coherence is given.

Implicits. Cochis [Schrijvers et al. 2019] is a calculus with highly expressive implicit resolution, including local instances. It achieves coherence by imposing restrictions on the implicit context and enforcing a deterministic resolution process. This allows for a much simpler coherence proof.

OCaml's modular implicits [White et al. 2014] do not enforce uniqueness of "instances" but dynamically ensure coherence by rejecting programs where there are multiple possible resolution derivations. This approach has not been formalized yet.

Other. Reynolds [Reynolds 1991] introduced the notion of coherence in the context of the Forsythe language's intersection types; he proved coherence directly in terms of the denotational semantics of the language.

In contrast, Bi et al. [2018, 2019] consider a setting where subtyping for intersection types is elaborated to coercions. Inspired by Biernacki and Polesiuk [2015], they use an approach based on contextual equivalence and logical relations, which has inspired us in turn. However, they do not create an intermediate language to avoid the problem of a more expressive target language. This leads to a notion of contextual equivalence that straddles two languages and complicates their proofs.

10 CONCLUSION

We have formally proven that type class resolution is coherent by means of logical relations and an intermediate language with explicit dictionaries. In future work we would like to mechanize the proof and adapt it to extensions such as quantified class constraints and GADT's.

ACKNOWLEDGMENTS

This work would not have been possible without the enlightening discussions with Dominique Devriese and George Karachalias. Furthermore, we would like to thank Alexander Vandenbroucke, Ruben Pieters and Steven Keuchel, as well as the anonymous ICFP 2019 and Haskell Symposium 2018 reviewers, for their constructive feedback. This research was partially supported by the Flemish Fund for Scientific Research (FWO) and the Hong Kong Research Grant Council projects number 17210617 and 17258816.

REFERENCES

Martín Abadi. 1999. *Protection in Programming-Language Translations.* Springer, 19–34.

Amal Ahmed. 2006. Step-indexed syntactic logical relations for recursive and quantified types. In *European Symposium on Programming (ESOP).*

Amal Ahmed. 2015. Logical Relations. https://www.cs.uoregon.edu/research/summerschool/summer15/curriculum.html.

Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. 2012. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming* 22, 2 (2012), 107–152.

Xuan Bi, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2018. The essence of nested composition. In *ECOOP.*

Xuan Bi, Ningning Xie, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. (2019).

Dariusz Biernacki and Piotr Polesiuk. 2015. Logical relations for coherence of effect subtyping. In *LIPIcs.*

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Haskell 2017.* ACM, 148–161.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

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

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover. (2015).

Dominique Devriese and Frank Piessens. 2011. On the Bright Side of Type Classes: Instance Arguments in Agda. In *ICFP '11*. ACM, 143–155.

Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. 2007. Modular Type Classes. In *POPL '07*. ACM, 63–70.

Phil Freeman. 2017. *PureScript by Example*. Leanpub. https://leanpub.com/purescript.

Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages* (Jan. 2019), 1–28. https://doi.org/10.1145/329031610.1145/3290316

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 (2006), 291–310.

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 (1996), 109–138.

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press.

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

M.P. Jones. 1993. *Coherence for qualified types*. Research Report YALEU/DCS/RR-989. Yale University, Dept. of Computer Science.

Mark P. Jones. 1994. *Qualified Types: Theory and Practice*. Cambridge University Press.

Mark P. Jones. 2000. Type Classes with Functional Dependencies. In *Programming Languages and Systems*. LNCS, Vol. 1782. Springer, 230–244.

Wolfram Kahl and Jan Scheffczyk. 2001. Named Instances for Haskell Type Classes. In *Proc. Haskell Workshop 2001*, Ralf Hinze (Ed.), Vol. 59.

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. *QuickChick: Property-Based Testing in Coq* (1st ed.). Software Foundations, Vol. 4.

Lex Spoon Martin Odersky and Bill Venners. 2008. Implicit Conversions and Parameters. In *Programming in Scala*. Chapter 21.

J. Garrett Morris. 2014. A simple semantics for Haskell overloading. In *Haskell 2014*, Wouter Swierstra (Ed.). ACM, 107–118.

James Hiram Morris Jr. 1969. *Lambda-calculus models of programming languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Team Mozilla Research. 2017. *The Rust Programming Language*. https://www.rust-lang.org/en-US/.

Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2017. Simplicitly: Foundations and Applications of Implicit Function Types. In *POPL '18*.

Dominic Orchard and Tom Schrijvers. 2010. Haskell Type Constraints Unleashed. In *Functional and Logic Programming*, Matthias Blume, Naoki Kobayashi, and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 56–71.

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

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-based Type Inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 50–61. https://doi.org/10.1145/1159803.1159811

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44.

Gordon Plotkin. 1973. *Lambda-definability and logical relations*. Edinburgh University.

John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *TACS '91*. Springer-Verlag, 675–700.

Tom Schrijvers, Bruno C.D.S. Oliveira, Philip Wadler, and Koar Marntirosian. 2019. COCHIS: Stable and coherent implicits. *Journal of Functional Programming* 29 (2019), e3. https://doi.org/10.1017/S0956796818000242

Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *TPHOLs '08*. Springer-Verlag, 278–293.

Richard Statman. 1985. Logical relations and the typed λ-calculus. *Information and Control* 65, 2-3 (1985), 85–97.

William W Tait. 1967. Intensional interpretations of functionals of finite type I. *The journal of symbolic logic* 32, 2 (1967), 198–212.

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *POPL '89*. ACM.

Leo White, Frédéric Bour, and Jeremy Yallop. 2014. Modular implicits. In *ML/OCaml 2014*.

Thomas Winant and Dominique Devriese. 2018. Coherent Explicit Dictionary Application for Haskell. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell (Haskell 2018)*. ACM, New York, NY, USA, 81–93.