Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Master Thesis Informatics

Type Class Coherence with Subtyping

Pascal Engel

Datum

Reviewers

Name Erstgutachter
(Informatik)
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Name Zweitgutachter
(Informatik)
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Engel, Pascal:
Type Class Coherence with Subtyping
Master Thesis Informatics
Eberhard Karls Universität Tübingen
Thesis period: von-bis

Abstract

In this work we aim to bring together three notions of polymorphism: Subtyping as a form of ad hoc polymorphism, parametric polymorphism and type classes. We do so by extending the research language `duo`, which already supports algebraic subtyping, with type classes. In order to ensure type class coherence we argue for a restrictive way of implementing instances for types that are in the subtyping relation.

Contents

Chapter 1

Introduction

In many functional programming languages such as Haskell, type classes are a powerful tool to generalize functions over different data types. This allows us e.g. to use the + operator both on `Int` and `Float` types. Another approach to overloading functions is subtyping, i.e. if a value of a certain type is expected we can also supply a value of a more specific type that is subsumed by the general type. For example, since the type of natural numbers `Nat` is a subtype of the integers `Int`, we can supply a value of type `Nat` for any function that expects an `Int`.

Although these approaches do not serve exactly the same purpose it is uncommon to find both concepts in the same language. In this work, I am going to show how it is possible to implement type classes in a language that supports subtyping. There are unique challenges when bringing both together because instances for certain types are going to be ambiguous.

The Haskell type `Either a b` roughly corresponds to the lattice type `a \/ b`. Given a type class `C :: * -> *` and instances `C a` and `C b` the instance `C (Either a b)` has to be defined by hand which can be easily done by pattern-matching and using the given instances. However, instances for lattice types are neither explicit nor straightforward: Since `a \/ b` does not have a uniquely-determined(?) constructor we might just implicitly derive `C (a \/ b)` from the given instances. However, this would make instances undecidable if we later on decide to implement an explicit instance of `C (a \/ b)`.

Together with subtyping we may be able to overload type classes even more. Consider

```
Show(if b then 42 : Nat else "Hello World" : String)
```

This term of type `Nat \/ String` appears to be well typed iff we can resolve the type class instances for `Show Nat` and `Show String`.

In the following we will explore how type classes interact with these and other lattice types.

Chapter 2

Polymorphism

In order to generalize functions over data types there have been several proposals to abstract over types in different programming languages: These can be summarised in three categories: parametric polymorphism, subtyping and ad-hoc polymorphism.

2.1 Parametric polymorphism

Parametric polymorphism is used in many functional languages but has been adopted as well in common main stream languages like Java. Using this feature, we ce can define functions without knowing the concrete representation of the arguments and result types. We can therefore implement abstract algorithms detached from concrete type representation. For example the universal identity function is typeable with `id : forall a. a -> a`.

This form of polymorphism allows us furthermore to reason about the behavior of functions: The `id` function mentioned above can only be implemented in one way:

```
id x = x
```

Its type enables us only to simply return the argument, because without knowing its type there is nothing else we could do with it. Similarily, for a function `g : forall a. a -> a -> a` there can be at most two implementations: One that returns the first and one that returns the second argument.

Many such *theorems for free* can be derived using parametric polymorphism, as Wadler has shown. [Wad89]

2.2 Subtyping

In many cases the specific semantics of types exhibit a hierarchy. In Object-oriented programming this hierarchy is given in the form of sub- and superclasses. We can express the relationship between super- and subclasses, or more generally super- and subtypes, in the form that all properties of the superclass is also exhibited in the subclass. [LW94]

In the case of OO-languages this means that, if the class `SubC` is a subclass of `SuperC`, then any method defined in `SuperC` is also going to be defined for objects of `SubC`. This enables us to use an object `SubC` wherever a `SuperC` is expected.

We denote $T \leq S$ for T is a subtype of S. Syntactically, this implies that if we have obatined the judgement $e : T$, we also have $e : S$. Therefore, we can use e in any context that expects the usage of a term of type S. Semantically, the subtyping relation can be understood analogously to sets in terms of the subset relationship $\subseteq$, meaning all terms e of type T are also of type S. [Rey98]

This permits many useful features in programming languages such as the reuse and abstraction of code to the supertypes and implicit coercions from a subtype to a supertype.

$$\frac{}{\tau :< \tau}\ \textsc{Refl}$$

$$\frac{\tau :< \sigma \quad \sigma :< \rho}{\tau :< \rho}\ \textsc{Trans}$$

$$\frac{\tau :< \sigma \vee \rho}{\tau :< \sigma}\ \textsc{Join}_1$$

$$\frac{\tau :< \sigma \vee \rho}{\tau :< \rho}\ \textsc{Join}_2$$

$$\frac{\tau :< \sigma}{\tau \wedge \rho :< \sigma}\ \textsc{Meet}_2$$

$$\frac{\tau :< \sigma}{\rho \wedge \tau :< \sigma}\ \textsc{Meet}_2$$

$$\frac{}{\tau :< \top}\ \textsc{Top}$$

$$\frac{}{\bot :< \tau}\ \textsc{Bot}$$

2.3 Ad-hoc polymorphism

In some - mostly imperative - languages it is possible to simply overload functions (e.g. we may define + both on integers and on float values, the correct implementation is then picked based on the argument type). However, there is usually no way to express this in the type system of these languages.

If our type system doesn't allow for ad-hoc polymorphism it may seem neccessary to write verbose code for basic function with respect to every concrete type it should be used for. An intuitive example for this (that is also the motivation for type classes in the original proposal) are arithmetic operators. We simply cannot define `(+) : Int -> Int -> Int` and then also `(+) : Float -> Float -> Float` in a different implementation, on which both types definetly rely on under the hood. But these types have nonetheless something in common. Namely that they both stand for *numerical* values, that hence support the usual arithmetic operations, like addition, multiplication, division and so on.

The idea of type classes is to generalise attributes of types with appropriate function. The class `Num` in Haskell expects that we can implement a number of numerical functions for a type τ if it is ought to be a member of the `Num` type class.

Shortened definition of the `Num` type class in Haskell. [1]

```
class  Num a  where
    (+), (-), (*) :: a -> a -> a
```

An instance would look like:

```
instance  Num Int  where
    (+) = intAdd
    (-) = intMinus
    (*) = intMul
```

In the end, this enables us to use elaborate concepts such as functors and monads to reason about programs.

[WB89]

2.4 Relation between different forms of Subtyping

The different forms of subtyping just discussed are not unrelated.

[1] As can be found in the default `Prelude`: https://hackage.haskell.org/package/base-4.16.2.0/docs/Prelude.html#t:Num

We can understand both parametric and ad-hoc polymorhism as a form of subtyping. E.g. the function `chooseSecond :: a -> Int -> Int` is a subtype of `const :: a -> b -> b`.

Similarily, the hierarchy of type classes expresses the same relationship exhibited by subtyping.

2.5 Coherence

Even though the general concept of type classes introduces a general meaning for each type class. The evaluation still strongly depends on implementation details found in specific instances. For example, for the `Ord` type class we may choose to implement the ordering in ascending or descending order. It is therefore crucial, that for each type the corresponding instance - if it exists - is uniquely determined by the type.

Reynolds [Rey91] describes the issue of 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.

For type classes, this means that no two instances should be able to be resolved for the same type. A rather obvious example would be to define two different instances for the same type. For example one instance of `Ord Int`, one with ascending and one with descending order. The ambiguity arises as soon as we make use of these instances and it is no longer clear which one should be picked for evaluation.

More surprisingly type class coherence is already violated for overlapping instances. As part of the standard prelude we find both `instance Show a => Show [a]` and `instance Show String` (with `String` being a type synonym for `[Char]`). The `Show` instance for Strings differs from the more general instance for lists.

Ambiguous programs should generally not be typeable. One example, also mentioned in the Haskell 98 report [Jon02] is this short program which simply reads a string to a data type and then converts it back to string without specifying which data type is being used:

```
f :: String -> String
f str = let x = read str in show x
```

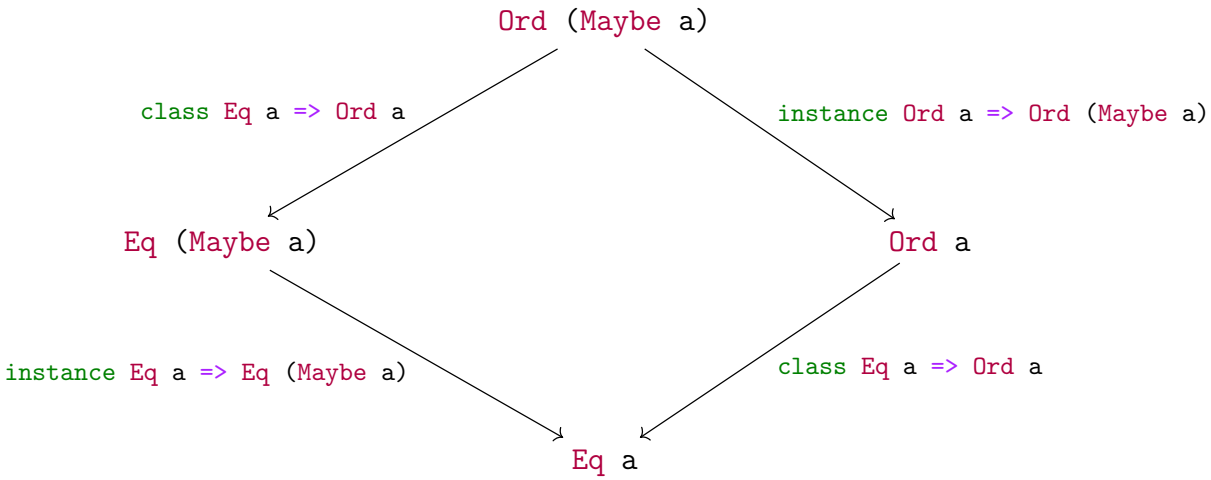
There may be multiple types that satisfy the type class constraints. The specific implementation of `show :: forall a. (Show a) => a -> String` and `read :: forall a.(Read a) -> String -> a` is therefore unknown.

In the Haskell98 standard, type class coherence is guaranteed by the syntactical equivalence of resolved instances. For each Haskell type there may be at most one instance defined for each type class.

2.5.1 Example

Consider superclasses in Haskell. E.g. the type class `Eq` is a superclass of `Ord`, written `Eq a => Ord a`. This means that for each type that we want to define an ordering for already needs to have equality defined on.

For example, given an instance for `Ord (Maybe a)` we can then derive that an instance for `Eq a` has to exist. As shown in the diagram, we can take different paths to do so but type class coherence guarantees that no matter which path we choose to resolve the instance, we will always find *the same* instance. In Haskell98 type class coherence guarantees that all such diagrams commute. So even if there may be different ways to resolve type class constraints, all of them preserve the same semantics.



Even though there are multiple ways to derive an instance for `Eq a` from an instance of `Ord (Maybe a)`, the derived instance has to be uniquely determined. Since the diagram commutes, there has to be exactly one instance for `Eq a`.

2.5.2 How to ensure coherence

There are several ways to ensure type class coherence. The easiest one can also be found in Haskell98. Here, overlapping or otherwise ambiguous instances are simply not allowed to be declared in the same namespace.

Another possibility somewhat alike ML-modules are named instances.

The latest proposal are instance chains which make the order in which instances are checked explicit. [MJ10]

Chapter 3

Type Inference

3.1 Type classes as predicates over types

Type classes are a way to *qualify* types for certain properties. E.g. we can say that for a type τ to fall under the `Show` type class means that τ is a showable type. This means every term $t : \tau$ has some property ϕ that is characteristic for Φ.

Using set-theoretic reasoning, we can motivate rules to reason about type classes by how properties are propagated through various type operators and the subtyping relationship. $\Phi(\tau)$ holds iff for every witness of τ some characteristic property ϕ of Φ holds. (Φ can stand for a type class, ϕ for a class method.)

Definition 1. *A* predicate over types Φ *is specified by a set of properties* ϕ_i. *For a type* τ *the application* $\Phi(\tau)$ *holds if and only if* ϕ *holds for every member of* τ.

Our syntax looks as follows:

$$
\begin{array}{lr}
\tau := \texttt{Nat} \mid \texttt{Int} \mid \ldots & \textit{Simple types} \\
\quad \top \mid \bot \mid \tau \wedge \tau \mid \tau \vee \tau & \textit{Lattice types} \\
\Phi := \texttt{Read} \mid \texttt{Eq} \mid \texttt{Show} \mid \ldots & \textit{Type classes} \\
\Xi := \Phi(\tau) \mid \Xi \Rightarrow \Phi(\tau) & \textit{Constraints}
\end{array}
$$

Let context $\Gamma = \{\Xi_1, \ldots, \Xi_n\}$ be a set of constraints (propositions).

The central rule motivated by Liskov substitution: If some property holds for σ and $\tau < \sigma$, If Φ is a property for σ and $\tau < \sigma$, then it should also be a property τ. Since all witnesses of τ are also witnesses of σ, the property Φ is propagated to τ.

$$\frac{\Gamma \vdash \Phi(\sigma) \quad \tau :< \sigma}{\Gamma \vdash \Phi(\tau)}\,\textsc{Sub}$$

If Φ is a property for both τ and σ, then it should also be a property of their union or join $\tau \vee \sigma$. Since all witnesses of $\tau \vee \sigma$ are witnesses of σ or τ, the property Φ holds for all witnesses of $\tau \vee \sigma$.

$$\frac{\Gamma \vdash \Phi(\sigma) \quad \Gamma \vdash \Phi(\tau)}{\Gamma \vdash \Phi(\tau \vee \sigma)}\,\textsc{Join}$$

If we know Φ for both τ and σ, the characteristic predicate also holds for all terms that are witnesses of both τ and σ, i.e. $\tau \wedge \sigma$. This rule is *admissable* because with $\tau :< \tau \wedge \sigma$ and $\sigma :< \tau \wedge \sigma$ we can derive the result using the Sub-rule.

$$\frac{\Gamma \vdash \Phi(\tau)}{\Gamma \vdash \Phi(\tau \wedge \sigma)}\,\textsc{Meet}$$

$$\frac{\Gamma \vdash \Phi(\sigma)}{\Gamma \vdash \Phi(\tau \wedge \sigma)}\,\textsc{Meet}$$

If Φ depends on Ψ, then for every τ with $\Phi(\tau)$ we also obtain $\Psi(\tau)$. For every witness of τ, the characteristic property ψ holds and because of $\Phi(\tau) \Rightarrow \Psi(\tau)$ we have ψ implies ϕ, ϕ holds for every witness of τ as well.

$$\frac{\Gamma \vdash \Psi(\tau) \quad \Gamma \vdash \Phi(\tau) \Rightarrow \Psi(\tau)}{\Gamma \vdash \Phi(\tau)}\,\textsc{Impl}$$

As a special case of Sub (since for any τ and σ we have $\tau < \tau \vee \sigma$ and $\sigma < \tau \vee \sigma$) we can also introduce elimination rules for unions:

$$\frac{\Gamma \vdash \Phi(\tau \vee \sigma)}{\Gamma \vdash \Phi(\tau)}\,\textsc{Join}$$

$$\frac{\Gamma \vdash \Phi(\tau \vee \sigma)}{\Gamma \vdash \Phi(\sigma)}\,\textsc{Join}$$

Axioms:

$$\frac{}{\Gamma, \Xi \vdash \Xi}\,\textsc{Axiom}$$

Every property holds for all witnesses of $\bot$, so every predicate trivially holds for $\bot$.

$$\frac{}{\Gamma \vdash \Phi(\bot)}\,\textsc{Bot}$$

3.2 Instance Resolution

With the Curry-Howard isomorphism we can interpret propositions as types and terms as proofs accordingly. Extending this notion we can interpret type classes as predicates over types. E.g. `Eq` σ would mean that the predicate (or property) of equality holds for type σ. We can provide a proof for this iff we can resolve an instance of `Eq` σ as a witness.

$$
\begin{aligned}
\tau &:= \texttt{Nat} \mid \texttt{Int} \mid \ldots && \textit{Simple types} \\
&\quad \top \mid \bot \mid \tau \wedge \tau \mid \tau \vee \tau && \textit{Lattice types} \\
\Phi &:= \texttt{Read} \mid \texttt{Eq} \mid \texttt{Show} \mid \ldots && \textit{Type classes}
\end{aligned}
$$

Notation:

- Γ: Context of known instances.
- Φ^+: Covariant type class Φ
- Φ^-: Contravariant type class Φ
- Rules annotated with * constrain the argument types to be equal, e.g. if we can infer $\texttt{Eq}(\tau \vee \sigma)$, both arguments of the class method `Eq` have to be either of type τ or σ.

Deriving instances:

$$\frac{\Gamma \vdash \Phi^+(\sigma) \quad \tau :< \sigma}{\Gamma \vdash \Phi^+(\tau)}\ \textsc{Sub}^+$$

$$\frac{\Gamma \vdash \Phi^-(\sigma) \quad \sigma :< \tau}{\Gamma \vdash \Phi^-(\tau)}\ \textsc{Sub}^-$$

Admissable rules:

Unclear yet, how this behaves for co-/contravariant type classes. Definetly problematic for `Eq`.

$$\frac{\Gamma \vdash \Phi^-(\sigma) \quad \Gamma \vdash \Phi^-(\tau)}{\Gamma \vdash \Phi^-(\tau \vee \sigma)}\ \textsc{Join}$$

This seems trivial, because we already have Sub. Additionally, given type class coherence the meet would always be $\bot$. So we do not gain anything from this rule.

$$\frac{\Gamma \vdash \Phi^+(\sigma) \quad \Gamma \vdash \Phi^+(\tau)}{\Gamma \vdash \Phi^+(\tau \wedge \sigma)}\ \textsc{Meet}$$

Axioms:

$$\frac{}{\Gamma, \Phi^+(\tau) \vdash \Phi^+(\tau)}\ \textsc{Axiom}^+$$

$$\frac{}{\Gamma, \Phi^-(\tau) \vdash \Phi^-(\tau)}\ \textsc{Axiom}^-$$

This seems trivial again:

$$\frac{}{\Gamma \vdash \Phi^+(\bot)}\ \textsc{Bot}$$

This seems very questionable (but does not entail obviously wrong things with $\textsc{Sub}^-$):

$$\frac{}{\Gamma \vdash \Phi^-(\top)}\ \textsc{Top}$$

Probably should not be a rule:

$$\frac{\texttt{instance}\ \Phi(\tau)}{\Phi(\tau) \vdash}\ \textsc{Decl}$$

3.3 Dictionary Passing

One typical way of implementing type classes is using *dicitonary passing style*. We use an isomorphism between type classes and records: E.g. the type class `Eq` defined as

```
class Eq a where
  eq :: a -> a -> Bool
  neq :: a -> a -> Bool
```

can be translated to a record type that preserves the structure of the class:

```
data DictEq a =
  DictEq { eq :: a -> a -> Bool,
           neq :: a -> a -> Bool }
```

Instances as witnesses of type classes can be translated accordingly to values of dictionaries. E.g.

```
instance Eq Int where
  eq = intEq
  neq = not . intEq
```

can be translated to a value of type `DictEq Int`:

```
intEqDict = DictEq { eq = intEq,
                     neq = not . intEq }
```

Here, the instances of a type class are carried in a dictionary that has to be passed as an additional argument to method calls. For example the term

```
show 5
```

would be compiled to

```
show showIntDict 5
```

where `showIntDict` is a dictionary that provides the relevant defintion of `show :: Int -> String` for the instance of `Show Int`. This allows us to "compile away" the overhead that type classes introduce to the surface language because type inference can fill in the correct dictionary that is ought to be used.

Generic constrained functions like

```
emphasize :: (Show a) => a -> String
emphasize x = show x ++ "!"
```

would simply pass around the dictionary and be compiled to something like:

```
emphasize :: ShowDict a -> a -> String
emphasize dict x = show dict x ++ "!"
```

[Kis21]

There is a drawback however, in the presence of subtyping it may not always be clear which dictionary is going to be needed for instance resolution. Consider the term:

```
show (if b then 42 :: Int else "Hello" :: String)
```

The inferred type of this expression should be `Int \/ String`. What we need here are essentially two dictionaries: One for `Int` and one for `String` because it is undecidable at compile time which dictionary is going to be used at runtime.

3.4 Intensional type analysis

Instead of passing around dictionaries, we can also resolve the relevant instance at runtime dispatching on the term's type. This requires our language to tag term with some representation of their type at runtime. In the before mentioned case, the definition of `show` could look something like this:

```
show t x = case t of
  Bool -> boolShow x
  Int -> intShow x
```

In the context of subtyping we would not require type equality but for `t` to be a subtype of some type for which an instance is defined. This also renders intensional type analysis more difficult: We can not simply match on a type (or its encoded representation [Wei00]) because we have to essentially solve the problem of subtyping between the type at hand and (at worst case) for each type for which an instance declaration is in scope. So, intensional type analysis may introduce a dramatic overhead in the presence of subtyping.

3.5 Type class coherence in the context of subtyping

Given `instance C a` and $sub < a$ and $a < sup$, we can neither have `instance C sub`, nor `instance C sup`.

Consider we have `Nat` :<`Int`. We can implement Monoid instances for both types. For natural numbers we choose multiplication as operator and accordingly 1 as neutral element. For integers on the other hand, we might prefer to choose addition as operator and 0 as neutral element, so we can expand to monoid to a group.

Building programs on top of these instances is going to get tedious as it will often occur that the more specific `Nat` type will be inferred, even if only want to deal with integers. Using the append operator exposed by the Monoid typeclass, therefore may lead to unexpected behavior.

In the simple arithmetic expression $(a \oplus b) \oplus c \oplus$ can have two different meanings based on the inferred types of a, b and c. Since type inferrence with subtyping is generally not quite obvious it may seem

Could we just use the most specific instance? This might have unexpected results. E.g. if we have `NonEmptyList < List`, we may not know during compilation whether `NonEmptyList` or `List` is being picked. Generally to infer the most specific type seems very hard. In this example filtering a `NonEmptyList` may or may not return an empty list and we may just have to assume that

$\text{show}_{a \vee b}$ x := if typeOf x == a then show_a x else show_b x

it is possibly empty. This may lead to hard to track behaviour when using overlapping instances.

A simple example for undecidable most specific instances can be better given with record types. If we need to resolve an instance for `C x : Int` and we already have instances for `C x : Int, y : Int` and `C x : Int, z : Int` neither instance is more specific than the other.

We should always check in an instance declaration whether this constraint globally holds.
To guarantee modularity we also have to check this for module imports (possibly hiding instances).

3.6 Implementation

There is a problem when we want to implement the union type.

Consider again the `Show` typeclass. It is intuitive to see how we can construct the `show` method for the union of two types, given that it is defined for both respecetively.

3.6.1 Multi Parameter Type Classes

The concrete implementation may impose further restrictions on instance resolution.

Our language differentiattes covariant and contravariant type classes. This distinction is made for the type variable declared in the class declaration.

In a covariant type class, type variables may only occur on covariant positions in the class methods signatures. A simple example for a covariant type class is the `Show` class:

```
class Show +a where
  show :: a -> String
```

Dually in a contravariant type class, type variables may only occur on contravariant positions in the class methods signatures. A simple example for a contravariant type class is the `Read` class:

```
class Read -a where
  read :: Read -> a
```

This distinction imposes a problem when we want to implement type classes in which the type variable may occur both in a covariant and contravariant position.

```
class Semigroup a where
  mappend :: a -> a -> a
```

One way to solve this problem is by distinguishing covariant and contravariant type variables. We can do so by introducing multi parameter type classes:

```
class Semigroup +a -b where
  mappend :: a -> a -> b
```

Instead of defining an instance for `Semigroup Nat`, we would then have to define an instance for `Semigroup Nat Nat`. The latter seems to be less intuitive because semigroups are not a relation between types but a property for just one type (the operator has to map two elements from one set into the same set). Since such cases may occur in many other classes (e.g. Num, Monad) it may be helpful to define syntactic sugar for type classes with mixed variance. Then, the less intuitive implementation as multi parameter type classes could be hidden on the surface syntax.

Discuss instance chains: We can relax this constraint by defining an explicit order in which instances should be picked/resolved. [MJ10]

Chapter 4

Summary

Chapter 5

Discussion and Outlook

Bibliography

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002.

[Kis21] Oleg Kiselyov. Implementing, and understanding type classes, 2021.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, nov 1994.

[MJ10] J Garrett Morris and Mark P Jones. Instance chains: type class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, 2010.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 675–700, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA, 1989. Association for Computing Machinery.

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. Association for Computing Machinery.

[Wei00] Stephanie Weirich. Encoding Intensional Type Analysis. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 92–106, 11 2000.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift