Maintainable Type Classes for Haskell

František Farka
University of Dundee
ffarka@dundee.ac.uk

Abstract

This paper addresses a long-term maintainability problem in Haskell type class system. In particular, we study a possibility of backward-compatible changes of existing class hierarchies. We summarize current proposed solutions to the problem and analyze their properties. Based on this analysis we derive our own language extension.

We discuss several possible applications of the language extension and compare the extension to other solutions. As a part of the paper we also give an implementation of the extension for the GHC compiler.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and TheorySyntax;; D.3.2 [*Programming Languages*]: Language Classifications Applicative (functional) languages;

Keywords Haskell, ad–hoc polymorphism, type class, instances, default superclass instances

1. Introduction

Haskell type classes are difficult to maintain. Haskell class declaration introduces a new data type class and operations – class methods – on it. The class declaration also may contain context that specifies superclasses of the class. A change in the class definition may case an error in compilation of source code that uses the class. We demonstrate this problem on somewhat simplified variants of two standard Haskell type classes, `Eq'` and `Ord'`. Consider a *Library* code provided by its *authors*, with following definitions of classes:

```
module Library where

class Eq' a where
        (≡) :: a → a → Bool

class Ord' a where
        (≤) :: a → a → Bool
```

Users of this Library can use these definitions in their own code, which we call *Client* code, by providing instances for the type classes. However it suffices to provide only instances of classes that are actually used:

```
module Client where

import Library

data Foo = ...

instance Ord' Foo where
  (≤) :: ...
```

When the authors of the Library realise that it is more convenient to make `Eq'` superclass of `Ord'` and changes the context of `Ord'`

```
class Eq' ⇒ Ord' a where
```

they introduce a breaking change—the original Client code does not compile with the new version of the Library and throws an error message:[1]

```
Client.hs:
    No instance for (Eq' Foo)
      arising from the superclasses of an instance
      declaration
    In the instance declaration for Ord' Foo
```

This issue can affect large volumes of code that uses old version of library. There is one notoriously known example in Haskell community: The *Functor–Applicative–Monad Proposal* (AMP) [8]. The `Monad` class is used heavily in Haskell source code. Before the proposal it was a standalone class without any superclasses. This proposal makes `Applicative` superclass of `Monad`. This change breaks any code that makes use of `Monad` and does not provide instances for both `Functor` and `Applicative`, for the reason we demonstrated in the case of `Eq'` – `Ord'` classes. However such breakage is undesirable and introducing this change took nearly 2 years [2, 12] and required a compiler support in form of additional warnings and manual adaptation of existing code. We describe this process in Section 2.2.

In this paper, we address a problem of introducing changes into class hierarchy in program development cycle and an impact of such changes on maintainability of existing source code. In particular, we claim that:

- some changes in class hierarchy are not backward compatible and break existing code;
- fixing these breakages requires long time and significant effort in terms of manual modifications of source code;
- an alternative way of introducing of changes into class hierarchy will make Haskell code more maintainable.

We give a means of introduction of such changes in backward compatible way.

[1] This example is produced with The Glorious Glasgow Haskell Compilation System, version 7.8.3

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Haskell Symposium 2015, September 3–4, 2015, Vancouver, Canada.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

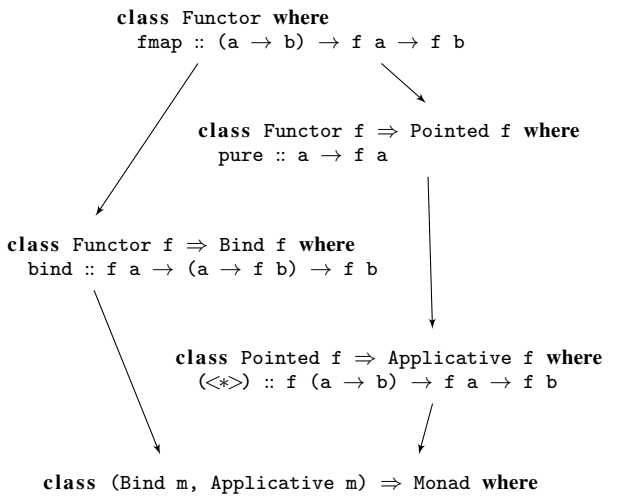

Figure 1. FPBAM class hierarchy

Haskell class hierarchy can be changed in several ways. In Section 2.1 we show that all changes can be decomposed into a small set of primitive changes. These primitive changes are in most cases either non-problematic by their their nature or can be dealt with using compiler *DEPRECATED* pragma. The only problematic change is adding a new superclass into the context of a class.

As an alternative to approach taken in case of AMP we propose to extend Haskell with a new language construct—*default superclass instance*. This construct allows author of a class definition to specify default instances of superclasses within class itself. These default instances are generated and used by compiler where proper instances are missing. With this new construct it is possible to make changes into class hierarchy, e. g. introduce new superclass constraints, instantly and without breaking of existing source code.

Beside the already mentioned examples of type class hierarchies – the simplistic `Eq'-Ord'` hierarchy and the `Functor`, `Applicative`, and `Monad` (FAM) hierarchy – we use two more hierarchies as our main examples. The first one is a direct extension of FAM hierarchy with `Bind` and `Pointed`[13] classes we refer to as FPBAM. This hierarchy is described in the Figure 1. The other one is the `Num` class and its possible enhancements.

1.1 Previous attempts to solve the problem

Several attempts have been made to address the problem of code breakage in a case of class hierarchy changes. In general we can distinguish two main trends: one focuses on type class definition and allows some form of default implementation of superclass method, e. g. [5, 4], and the other groups several classes into an alias and allows to use this alias in definition of an instance [18, 4]. We analyse these approaches in Section 3. Default superclass instances proposal gives a suitable syntax but fails to address proper semantics of the extension. It is not obvious how to deal with cases of multiple default instances. Consider the diamond in *Library* in Figure 2 and *Client* that provides instances for classes `B`, `C`, and `D`. It is not obvious what should happen – which instance should be chosen – it is posiible to provide an instnace from definition in either of the classes `B`, `C`, and `D`. Class aliases suffer with similar problem where it is not obvious how should methods be distributed from instance of the alias into instances of partial classes that the alias consists of.

Conor McBride has provided [16] implementation of superclass default instances as a part of his Strathclyde Haskell Enhancement.

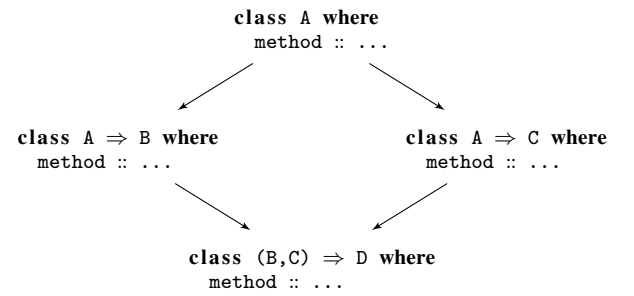

Figure 2. Diamond class hierarchy

However this implementation was carried out as Haskell preprocessor and is limited to the scope of a single file. This implementation also does not address issue of multiple default instances.

1.2 Contributions

In this paper we give a solution to the problem stated in Section 2 – a problem of introduction of changes in class hierarchy while keeping the hierarchy backward compatible with existing code that uses it. Our analysis of previous attempts in Section 3 allows us to devise a coherent solution that allows library authors to make changes in their source code without breaking the existing code that uses the library. This makes the code–base more maintainable in the long perspective. In particular, we

- propose a new language extension – *Superclass Default instances*;
- provide a formal description of the extension;
- discuss interaction of our extension with other language extensions; and
- demonstrate our solution on several use-cases, and discuss further applications of our extension—e. g. as a mean of providing simplified class abstraction.

We give a formal description of syntax of the extension in the style of Haskell 2010: Language Report[14] and provide detailed semantics. We specify the selection mechanism in the case of multiple default instance declarations in Section 4.1. This section also describes benefits of this selection mechanism and addresses behavior of compiler notifications and warnings. The GHC offers a variety of language extension. Some of these extensions involve type class system and may conflict with our extensions, e. g., *Multi–Parameter TypeClasses*, *Undecidable Instances*, or different extensions to deriving mechanism. Possible conflicts with other language extensions are analysed in Section 5.

We also discuss applications of our language extensions in other situations, as a mean of providing simplified class abstraction with more elaborate class hierarchies. A brief overview of our implementation of described extension is given in Section 6.

The source code of our implementation is available at: `https://github.com/frantisekfarka/ghc-dsi`.

2. Maintainability Problem

This section describes a problem of the current type classes design. Type classes are one of the core features of the language [10]. However, any change in a type class hierarchy currently requires rewriting the appropriate instance implementations. Therefore any change to the hierarchy breaks backward compatibility and thus poses significant problem to maintainability of source code.

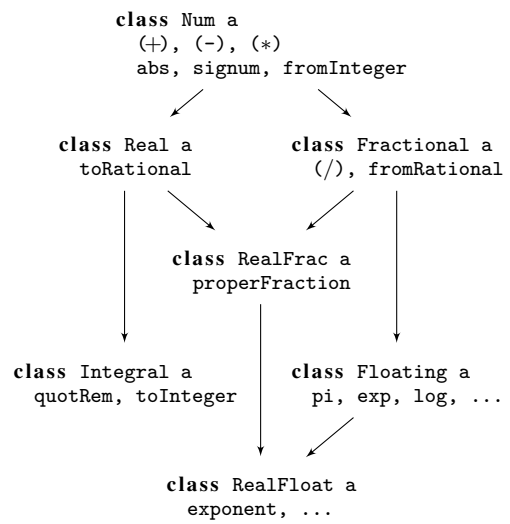

Figure 3. Standard Numeric Classes and Related Operations

It is often desirable to change class hierarchy. In some cases the proper class relation is not understood at first and a superclass of some class is missing as is the case with `Monad` class as we show in Section 2.2. In other situations new concepts emerge and a class hierarchy needs to be refactored. John Wiegly proposed [23] to add a `Semigrupoid` class as a superclass of `Monoid` class. In the subsequent discussion Edward Kmett pointed out that this change would break existing code due to missing instances of `Semigrupoid` where the instance of `Monoid` already exists.

Beside changes in existing class hierarchy there are some situations where is the behavior of superclass instance uniquely determined. Such example is the `Traversable` class [3]. Instances of it's superclasses `Functor` and `Foldable` should satisfy:

```
fmap f = getId ∘ traverse (Id ∘ f)
foldMap f = getConst ∘ traverse (Const ∘ f)
```

where `Id` and `Const` are identity applicative functor and constant applicative funcor respectively. Although the behavior is documented, the instances still has to be written manually and may result in erroneous code in case hand written instances violates the expected laws. Another of standing problems regarding existing classes is the hierarchy of the Standard Numeric Classes and Related Operations [14]. Figure 2 shows this hierarchy.

Dylan Thurston et al. [7] have argued that the standard set of numeric classes is limited in extensibility and violates semantics of operations. The hierarchy is rather complex for novice user and still does not provide means for expressing of algebraic notions, e. g. notion of commutative group. However, the change in the hierarchy of these classes would break existing code.

2.1 Altering Type Class Hierarchy

Previous section did not restrict changes in a class hierarchy in any way. Various such changes may occur. Nevertheless, this section shows that any change can be decomposed into a series of six elementary actions. Consider following scenarios on an arbitrary class hierarchy:

- **Change context of** an existing **class**, i. e. superclasses that the class is dependent on.
- **Add** a new **class** to the hierarchy. The class may be either standalone or connected with the rest of the hierarchy **as a subclass or a superclass**.
- **Remove** an existing **class** from the hierarchy. Yet again the class may be either standalone or connected to the hierarchy.
- **Refactor** existing **class** hierarchy by either merging more existing classes into one or by dividing an existing class into more new classes. The new classes may be in various relations as subclasses and superclasses.
- **Move** a method from a subclass to a superclass or the other way around **or add** a new class **method**.

Our simplistic `Eq' – Ord'` example is an instance of the first scenario. John Wiegly's `Semigrupoid` is an instance of second scenario. The Introduction have given a real world example of the first scenario—the FAM proposal. If the Standard Numeric Classes were to be redesigned scenarios three to five would take place in this process.

These are complex scenarios. However, one can see certain actions take place repeatedly. E. g. in the case of `Semigrupoid` new class definition has to be provided and this class must be added into the context of `Monoid`. We define a set of these actions.

Definition 1 (Set of elementary actions $\mathfrak{A}$)**.** *Let $\mathfrak{A}$ be a set of following actions:*

a_1 *add a class without methods and superclasses;*
a_2 *remove a class without methods and superclasses;*
a_3 *make an existing class a superclass of another existing class;*
a_4 *remove a superclass constraint from an existing class;*
a_5 *add a new method to some class;*
a_6 *remove an existing method from some class.*

We call the actions from set $\mathfrak{A}$ *elementary*. Clearly it is possible to compose above stated scenarios from the set of actions $\mathfrak{A}$. Assume the second scenario—add a class as a superclass or subclass . By using action a_1 for the class, action a_5 repeatedly for any method of the class and action a_3 for any superclass or any class, to which it is a superclass, we get the desired change in hierarchy. We deem compositions for the rest of scenarios similarly straightforward and these compositions are omitted for brevity. The observation on compositions allows us to state a following theorem:

Theorem 1 (Altering a Haskell class hierarchy)**.** *It is possible to compose any change in a Haskell class hierarchy from actions from the set $\mathfrak{A}$*

Proof. We model a Haskell class hierarchy as a directed acyclic graph with a set of labels for every vertex where classes are vertices, superclass relations are edges from subclass vertex to superclass vertex, and methods are labels on vertices.

Assume two arbitrary hierarchies ι_1 and ι_2. Assume the notation $V(\iota_i)$, $E(\iota_i)$, and $L(\iota_i)$ for a set of vertices, edges and labels of vertices respectively.

Then use actions from $\mathfrak{A}$ on ι_1; use a_6 on $L(\iota_1) \setminus L(\iota_2)$ to remove labels, use a_4 on $E(\iota_1) \setminus E(\iota_2)$ to remove superclass constraints and use a_2 on $V(\iota_1) \setminus V(\iota_2)$ to remove classes; use a_1 on $V(\iota_1) \setminus V(\iota_2)$ to add new classes, use a_3 on $E(\iota_1) \setminus E(\iota_2)$ to add constraints, and use a_5 on $L(\iota_1) \setminus L(\iota_2)$ to add labels. This sequence of operations transforms the graph ι_1 into the graph ι_2. □

These actions are involved with different problems when introduced to existing library. In particular these are:

a_1 **– add a class without method and superclasses** This action does not introduce any problems but the one problem common to all programming languages—new class name may clash with other identifier. We do not provide any solution for this particular issue and simply assume it can be avoided by careful

selection of identifiers by both author of the library and author of source code that uses the library.

a_2 **– remove a class without methods and superclasses** Class can be removed from library only when it is not used by any source code that we want to keep backward compatible with the library. Otherwise we get "Not in scope" error. Such class can be attributed with *DEPRECATED* pragma and its removing from library delayed. In this way, all the new code received compile time warning that this class is not intended to be used any more and all the existing code still can be compiled.

a_3 **– make an existing class a superclass of another existing class** This change causes the error that was demonstrated on `Eq'` – `Ord'` example in the Introduction of this paper—if some source code provides instance only for `Ord'` it fails to compile with the new version of the library. This is a serious limitation, e. g., it prevents of direct introduction of some abstraction. We have already mentioned the `Applicative` as a superclass of `Monad` and `Semigrupoid` as a superclass of `Monoid`.

a_4 **– remove a superclass constraint from an existing class** This change does not cause any problems. Some instances from before the change may be superfluous as these are no longer required by superclass constraint.

a_5 **– add a new method to some class** The new method causes a problem only when its name clashes with existing method in user code causing ambiguous occurrences. This is similar to the problem causes by action a_2. We assume it can be avoided by careful identifier selection.

a_6 **– remove an existing method from some class** Existing method cannot be removed without code breakage. This is similar to action a_2 and can be dealt with by DEPRECATED pragma in similar manner.

This list shows that only problematic change in class hierarchy is introducing new superclass constraint. Other can in our opinion be dealt with by careful section of identifiers and by DEPRECATED pragma. One can oblige that semantics of moving a method from one class to another is different from removing a method from one class and adding a new method to another class. However, the result of both is structurally identical. We deal with semantics of moving of a method in Section 4.

There are currently two ways of introducing superclass constraint. We briefly describe both of these.

2.2 Deprecation with Compiler Support

Haskell does heavily use the `Monad` class—it is tightly incorporated into the language through the `do` notation. Although it is known that every `Monad` is, in principle, a subclass of `Applicative` [8], it was not true so in the standard Prelude before GHC 7.10. Changing the `Monad` class into subclass of `Applicative` breaks any client code that uses the class and does not instantiate `Applicative`.

A three phase process was adopted by GHC in order to avoid it. In the first phase, the compiler was patched such that it issues warnings in case of missing instances [2] of `Applicative` and `Functor` where an instance of `Monad` exist. In the second phase all authors were expected to fix their code using the new compiler warnings. In the third phase the change was applied [12].

The three phase process illustrates what is in our opinion the major problem of any considerable changes in type class hierarchy: it is a long lasting process that requires a vast amount of manual work and still can result in broken source code as in case of the AMP and code that failed to provide an instance of `Applicative` in the phase two.

This procedure is in principle – with some generalised support of deprecation of missing instances from a compiler – possible with any such change in a class hierarchy. Though, it requires manual effort and some transitional period of time for users of the affected library to implement missing instances.

2.3 Subclass to Superclass Instance

A superclass instance may be provided, in some cases, from a subclass instance when we allow the *Flexible Instances* and *Undecidable Instances*. Assume following simplified classes `Functor'` and `Monad'`:

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}

class Functor' f where
        fmap' :: (a → b) → f a → f b

class Functor' m ⇒ Monad' m where
        return' :: a → m a
        (>>=) :: m a → (a → m b) → m b
```

With relaxes conditions on instances of the language extension following is a valid instance:

```
instance Monad' m ⇒ Functor' m where
        fmap' f a = a >>= (return' ∘ f)
```

Now the client code may instantiate only the `Monad` class:

```
data Id a = Id a deriving (Show)

instance Monad' Id where
        return' a = Id a
        (Id a) >>= f = f a

useFmap = fmap' (+1) (Id 5)
```

However, the GHC documentation [9] describes some problems that may arise with the two extensions, e. g., compiler may not be able to resolve the instance.

2.4 Design Goals of a Solution

We have described a problem with the current design of Haskell type classes and presented different solutions that are available. We believe that neither of the solutions addresses the problem in a sufficient manner.

The first approach (2.2) is not straightforward and requires manual adaptation of existing code. The second approach (2.3) requires some additional language extension that may cause compiler to reject the instances.

Based on these observations we state that any approach that aims to solve above given problem should address following goals:

- The approach should allow changes to a class hierarchy. In particular adding new classes and changing structure of existing hierarchies should be possible.
- The approach should not break any existing code, i. e., the changes should be backward compatible.
- The approach should not introduce any problems to the program compilation, e. g. undecidability of instances.

These design goals lead us to proposing a new language extension in Section 4.

3. Previous Work

This section addresses previous work and points out particular shortcomings. Three main lines of ideas can be identified in previous work:

Default instances as described in [5] and [16]

Default method implementation as described in [4].

Class aliases as described in [18], [21], and [4]

3.1 Default Instances

GHC proposes [5] a language extension that enables programmer write implementations of subclasses that imply their superclass implementations. Using our `Eq' – Ord'` example the proposed syntax is:

```
class Eq' a ⇒ Ord' a where
        (≤) :: a → a → Bool

        instance Eq' a where
                x ≡ y = x ≤ y && y ≤ x
```

Give this language construct default instance for `Eq'` is provided by compiler. The proposal requires each default superclass instance to be an instance of a different class. The proposal also discusses an opt-out mechanism to prevent generating of a default instance. It is done with a syntactic construct **hiding** as follows:

```
instance Ord' a where
        ...
        hiding instance Eq'
```

This construct allows user defined instance of *Super*. The other discussed variant is a quiet exclusion policy with following variants of dealing with intrinsic instances and explicit instance clashes:

- rejecting duplicate instance declaration
- allowing explicit instance supersede intrinsic default with a warning
- allowing explicit instance supersede intrinsic default without any warning

Second variant is considered by authors of the original proposal to be a pragmatic choice. However, proposal fails to recognize that there can be multiple default instances. Assume that following class is added to the hierarchy:

```
class Eq' ⇒ Dro' a where
        (≥) :: a → a → Bool

        instance Eq' a where
                x ≡ y = x ≥ y && y ≥ x
```

When using (≡) on a data type for which user provided both instances of `Ord'` and `Dro'` it is not obvious which default instance to use. There are two candidates—one instance generated in `Ord'` and other in `Dro'`. We consider a pragmatic choice to select such instance that is:

- Common to all paths from class where is the method introduced to any class for which is the ordinary instance provided.
- Is last among such instances when we consider topological order on the directed acyclic graph of the hierarchy in *Library*.

The rationale behind the first rule is to select an instance among all the candidates that is consistent. A subclass is viewed as a specialization of a superclass and thus the common ancestor is seen as an abstract enough class to provide implementation sufficing all candidates. The second rules adheres to this specialization in the manner that a subclass has more specific information on the data for which is the instance provided and thus can be implemented in a more efficient way. Under such assumptions it is reasonable to choose the candidate for default instance which originates later in the hierarchy.

These rules are generally not guaranteed to result in selection of a single candidate. We discuss this issue further in our proposal below.

3.2 Default Method Implementation

Class System Extension Proposal presented in [4] endorses the idea of providing default implementation of any ancestor method directly in class. This approach may derive suitable method definition from subclass instance:

```
class Eq' a ⇒ Ord' a where
        (≤) :: a → a → Bool

        a ≡ b = (a ≤ b) && (b ≤ a)
```

The behavior of method implementation is specified as:

- Class and instance declarations allow implementation of any method in a class or any superclass.
- Whenever an instance declaration is visible, there is always a full set of instance declarations for all superclasses. This is done by supplementing the set of explicitly given visible instance declarations by automatically generated implicit instance declarations.
- The extension proposes the policy of the most specific method implementation. This means using explicit instance over the default one and using subclass method over superclass method.
- Modules export only specific instance declarations.

Proposal observes that the resolution of an overloaded method depends on the visible instances in the module where method is called. Therefore, overloading needs to be resolved before merging the modules together, in particular inlined method overloading needs to be resolved before the method is inlined.

Proposal also indicates that with the aforementioned changes a compiler has to consider all the predicates in the context to determine the source of the overloaded function, whereas now it is sufficient to look only for particular instance.

There is a certain issue with this approach. Assume following library code which defines classes `Eq'`, `Ord'`, and `Bounded'`:

```
class Eq a where
        (≡) :: a → a → Bool

class Eq' a ⇒ Ord' a where
        (≤) :: a → a → Bool

class Bounded' a where
        minBound, maxBound :: a
```

And a client code which uses the class `Bounded'`:

```
data MyData = ...

instance Bounded' MyData where
        minBound = ...
        maxBound = ...
```

If we want to add the class context of `Ord'` to the class `Bounded'` in such manner this change does not break the existing client code we need to provide instance of `Ord' MyData` and `Eq' MyData` transitively. The solution provided in [4] is to always generate automatically full set of instance declarations for all superclasses. This solution does not deal with the problem. Assume that we the alter the class definition in the following manner:

```
class Eq' a where
        (≡) :: a → a → Bool

class Eq' a ⇒ Ord' a where
        (≤) :: a → a → Bool

        -- default method for Eq'.(≡)
        x ≡ y = ...
```

```
class Ord' a ⇒ Bounded' a where
        minBound, maxBound :: a

        -- default method for Eq'.(≡)
        x ≡ y = ...
```

When using this version of the module providing that default instances are generated it is not obvious which version of $\equiv$ should be used. Possible options are:

- compilation results in an error due to ambiguous occurrence of the method
- compilation uses some default policy for which method to use

The first option is not favorable as it result in code breakage. The second option requires reasonable policy for selecting one implementation among all candidates even in case of more elaborate class hierarchies than the hierarchy in the example.

3.3 Class Aliases

Class aliases are presented in several different proposals [18, 4, 21]. Aliases are proposed both as a mean of providing default method implementation and a single name for multiple classes. In order to maintain backward compatibility class aliases are usefull in combination with default superclass methods. This problem is hinted in proposal [21]. Assume following classes `Applicative` and `Monad`:

```
class Functor f ⇒  Applicative f where
        pure    :: a → f a
        (<*>)   :: f (a → b) → f a → f b

class Monad m where
        return  :: a → m a
        (>>=)   :: m a → (a → m b) → m b
```

These two classes are independent of each other and it is sufficient when using them in client code to define instance just for `Monad`:

```
data MyData =○ ..

instance Monad MyData where
        m >>= k          = ...
        return k         = ...
```

Using class alias as a mean of providing default implementation is in this case problematic. Assume we want to add `Applicative` into the context of `Monad` in such way the change is backward compatible. In this case we need to provide new class e. g. `Monad_` which implements this change and keep `Monad` as an alias for both `Applicative` and new `Monad_` classes:

```
class Applicative m ⇒ Monad_ m where
        return  :: a → m a
        (>>=)   :: m a → (a → m b) → m b

class alias Monad m = (Applicative m, Monad_ m) where
        fmap    :: (a → b) → m a → m b
        fmap f ma =  ma >>= (λa → return (f a))

        pure    :: a → m a
        pure a = return a

        (<*>)   :: m (a → b) → m a → m b
        mf <*> ma = mf >>= λf → ma >>= λa → return (f a)
```

This works fine considering original code which uses `Monad` and does not instantiate `Applicative`. However we now have two different names for `Monad` and it is not obvious which one should user who is aware of their existence use. The class aliases in this case help to remove amount of type classes held in hierarchy for backward compatibiliy.

On the other hand class aliases are useful when dividing class into two new. Assume that we want to subdivide class `Monad` from previous example into classes `Pointed` and `Bind`:

```
class Pointed a where
        return  :: a → f a

class Bind a where
        (>>=)   :: m a → (a → m b) → m b
```

In this case we do not need class `Monad` – there is no method which should this class contain. Instead class alias provides both backward compatibility and is handy to use with new code that is aware of `Pointed` and `Bind`:

```
class alias Monad m = (Bind m, Pointed m)
```

The other significant feature of class alias is that it enables programmer to provide single instance that gives declarations of methods of multiple classes. This in turn allows instantiation without the necessity of renaming of the methods. Assume two related classes `LowerBounded` and `UpperBounded` and a class alias `Bounded`:

```
class LowerBounded a where
    minBound :: ...

class UpperBounded a where
    maxBound :: ...

class alias Bounded a = (LowerBounded a,
    UpperBounded a)

instance Bounded MyData where
    minBound = ...
    maxBound = ...
```

The two instances for alias are generated accordingly. Without the alias functionality the `Bounded` must by a class with superclasses `LowerBounded` and `UpperBounded`. The instances for these two may be defaulted, but methods `minBound` and `maxBound` must be renamed it is not possible to have a method of same name in class and its superclass.

4. Language Extension Proposal

In this section we give a proposal of new language extension that solves problem described in Section 2. Based on and observations in the previous section we derive our own proposal which aims to address issues described in the Section 2.1, i. e.:

- Add or remove a superclass into the context of a class without breaking a client code, where is the class already use. Client code may or may not instantiate the superclass.
- Create a new class or remove an existing class that is not either a superclass or a subclass of any other class.
- Move a method from a class into either a superclass or a subclass.
- Add or remove a class method.

In the Section 2.1 we argued that actions from $\mathfrak{A}$ are sufficient enough to compose arbitrary change in the hierarchy. In this section we describe possible issues with such actions on existing code and discuss effects of the change when writing a new code.

When adding a superclass into a context of subclass – action a_3 – we distinguish several situations. Assume two classes, a class `Foo` and `Bar`, that were not (indirect) subclass of each other and we want to add the `Bar` into the context of `Foo` such that `Foo` is a

subclass of the class `Bar`. Assume that there is already a client code where `Foo` is brought into the scope and then subsequently used, i. e. some instance is provided. Then if there is also an instance of `Bar` in the client code everything works fine.

On the other hand when the instance of `Bar` is missing compilation results in an error message similar to:

```
No instance for (Foo MyData)
  arising from the superclasses of an instance
declaration
Possible fix: add an instance declaration for
(Foo MyData)
In the instance declaration for ‘Bar MyData’
```

Also without an instance any method of `Bar` is guaranteed not to be used in this scope, thus there can be class methods or functions with the same local name and occurrences of these methods can become ambiguous if the class `Bar` was not in the scope before the change in the hierarchy. The possible fix to this issue is to provide an implicit instance of `Bar` as we discussed earlier. However, this approach have several problems of its own. First, what to do with instance methods. We can either

- provide a default implementation in the class definition, or
- not to provide any definitions.

The first approach requires an active change in the class definition whereas later results in an error when method without definition is used. The last situation which can occur when adding the class `Bar` into the context of the class `Foo` is that only the class `Bar` is used in a client code. In this situation the client code works without any problems.

Assume the converse situation, i. e. there is a superclass `Bar`, its subclass `Foo`, and we want to remove the context of `Bar` from `Foo`, i. e. the action a_4. This action does not result in any problems with respect to the old client code.

The action a_1 of adding a new class that is neither superclass nor subclass of any other class also does not bring any problems beside the obvious issues with the name clashes. It is possible to minimize such issues by placing new class into separate submodule. Conservative treatment of export list of the module can also mitigate some problems.

Removing of an existing class – action a_2 – that is neither superclass not subclass of any other class is possible as long as the class is not used. In the case the class is rendered obsolete it should be deprecated by a pragma mechanism [9]. But in cases where the class is to be deleted as a result of the other action discussed, e. g. the class was split into two, it is necessary to provide it both for backward compatibility and for use in new code. It is possible to provide a class of the same name with the classes in which it was split as superclasses and default implementation original methods– which are now in superclasses. The other way is to provide a class alias of the same name as the original class aliasing the classes in which it was split. We consider the second approach superior as it result in shorter, more readable code.

Adding a method to a class – action a_5 – does not cause a problem in current Haskell. Such action results in an incomplete instance in the code where the class is instantiated and consecutively in compiler warning. Nevertheless, the compiled code works as expected. Removing a method – action a_6 – represents a similar problem to the removing a class scenario. Yet again we prefer deprecation of the method over deleting.

When moving a method from a superclass to a subclass or the other way around it, which is a composition of actions a_5 and a_6, is important to distinguish whether the change occurs also with the change in class hierarchy, whether the method was moved into superclass. In such case the method implementation for the subclass can be provided in the default instance. In the other case there can be an old code expecting original layout of method in classes and the change is not possible. In our opinion this case is better solved by deprecation pragmas.

topdecl	→	`class` [*scontext* `=>`] *tycls tyvar* [`where` *cdecls*]
scontext	→	*simpleclass*
	|	`(` $simpleclass_1$ `,` ... `,` $simpleclass_n$ `)` $(n \geq 0)$
simpleclass	→	*qtycls tyvar*
cdecls	→	`{` $cdecl_1$ `;` ... `;` $cdecl_n$ `}` $(n \geq 0)$
cdecl	→	*gendecl*
	|	(*funlhs* | *var*) *rhs*
	|	`default instance` ***qtycls dinst*** [`where` ***didecls***]
dinst	→	*gtycon*
	|	`(` $gtycon\ tyvar_1\ \ldots\ tyvar_k$ `)` $(k \geq 0,\ tyvars$ distinct$)$
	|	`(` $tyvar_1$ `,` ... `,` $tyvar_k$ `)` $(k \geq 2,\ tyvars$ distinct$)$
	|	`[` *tyvar* `]`
	|	`(` $tyvar_1$ `->` $tyvar_2$ `)` ($tyvar_1$ and $tyvar_2$ distinct)
didecls	→	`{` $didecl_1$ `;` ... `;` $didecl_n$ `}` $(n \geq 0)$
didecl	→	(*funlhs* | *var*) *rhs*
	|	(*empty*)

Figure 4. Extended Class declarations with Default Superclass Instance

Based on these possible situations we want to devise a new language extension proposal. With accordance to the design goals we specified in the Section 2.1 we consider most significant the changes in the class hierarchy. We consider an extension that enables programmer to make such changes a significant benefit to the maintainability of any existing codebase. Problems which are not ceased by changes in hierarchy can be in our opinion solved by careful choice of identifier names and export lists.

4.1 Superclass Default Instances

We propose to add a new syntax construct into the class definition as described in the [14]. Programmer may provide a *default superclass instance* in the class method for any of it's superclasses, e. g.:

```
class Functor f ⇒ Applicative f where
        pure :: a → f a
        (<*>) :: f (a → b) → f a → f b

        default instance Functor f where
                fmap :: (a → b) → f a → f b
                fmap f x = pure f <*> x
```

For any instance of `Applicative` the compiler generates the implicit default superclass instance of the class `Functor`. This instance is used when there is no ordinary instance of `Functor`. When there is both superclass default instance and ordinary instance the later is used. This behavior and a selection mechanism among multiple default instances is formally described below. An instance of the class does not change in any manner.

Haskell Language Report 2010 [14] describes formal syntax of type class declaration. With respect to the proposed change we adjust formal syntax as shown in the Figure 4. This adjustment

allows a new language contruct—a class declaration now contains nested default instance declaration of the general form:

```
class cx ⇒ D u where
      default instance C u where cdecls
```

This introduces new *default superclass instance* of the class C. The class `C` must be an (indirect) superclass of D. The nested default superclass instance declaration rules *dinst*, *didecls*, and *didecl* respects syntactic structure of instance declaration and restrictions on instance declarations hold accordingly. In particular u must take a form of a type constructor to simple type variables $u_1, \ldots, u_n$., type constructor must not by a type synonym and u_i must be all distinct.

The context is more complicated. A context of ordinary instance is expressed through set of classes and superclasses of these classes are present implicitly in this context due to the behavior of instance declarations. The context of a default instance contains only the class of its declaration, all indirect superclasses are included implicitly, but without all classes for which is any default instance declaration present in the class. Assume that dis is a set of classes that are being provided with default instance in the class `D` and that `a` id the type variable of the class. The context di_i of each instance from dis is:

$$di_i = D\ a$$

This allows programmer to use any method from class or superclass but those being defined in default instances, and allows instance resolving.

The declaration of default superclass instance may contain binding only for the class methods of C. If no binding is given for some method default method in the class declaration is used. If there is no such method the method of the instance is bound to *undefined*. The declaration of a default instance does not contain any signatures or fixity declarations. These were provided in the superclass that is instantiated.

We do not change the declaration of ordinary instance. However there can be both ordinary instance and either one or multiple *default instances* in the same scope. There cannot be more ordinary instances due to overlapping instances restriction. We consider the graph of all classes in scope. This graph is required to be acyclic [14] and has natural ordering $\prec'$ generated by class dependencies.

Definition 2. *For C, D classes let* $\prec'$ *be*

$$C \prec' C$$

and

$$C \prec' D \leftrightarrow \textit{C is an immediate superclass of D}$$

We introduce ordering on all classes $\prec$ as a transitive closure of $\prec'$:

Definition 3 (Class ordering)**.** *For A, B classes let* $\prec$ *be*

$$A \prec C \iff A \prec' B \vee \exists C : A \prec' C \wedge C \prec B$$

Assume the set consisting of ordinary instance (if there is one) and all *superclass default instances*. We call this a set of candidate instances or *candidates* for short. The instance selection mechanism applies following rules on the set of *candidates*:

Rule 1 If there is an ordinary instance select this instance.

Rule 2 Select all instances $I_i\,a$ such that for any other instance $I_n\,a$ holds

$$I_i\ a \prec I_n\ a \ \text{ or } \ I_n\ a \prec I_i\ a$$

Rule 3 Select an instance $I_i\,a$ such that for any other instance $I_n\,a$ holds

$$I_n\ a \prec I_i\ a$$

The motivation for the Rule 1 is to preserve backward compatible behavior, i. e. to select existing instances, and to enable user to provide his own implementation of the instance. We propose to issue a warning when a proper instance is selected over a default instance.

The motivation for the Rule 2 is to decide between two instances that are not superclass of each other. In this case we omit them both and try to select their common ancestor. We expect the ancestor to be general–or abstract–enough to provide sufficient default instance.

In the Rule 3 we have possibly several instances that are all either a superclass or a subclass of each other. The rationale behind the rule is to select the instance which is the least abstract, i. e. the most specific. We expect this instance to provide possibly better implementation regarding the performance as it has the most specific problem related information.

5. Relation to the Language Platform

In this section we put our work into broader perspective. We discuss several topics that are not necessarily mutually related but each of the topics is directly connected to our proposal. In particular we focus on a relation to other language extension and applications of our work to the actual language platform.

5.1 Relation to the Existing Language Extensions

The Haskell language consists of extensions not specified in [14]. The GHC, upon which we build our implementation (see Section 6), contains variety of additional language extensions that can be looked up in [9].

We consider important to assess possible clashes with other extensions even though proper testing should be provided. Nevertheless, such testing is extremely time consuming due to the great number of the extensions[2]. In sight of this fact we do not list all the extension but only the extension we evaluated as possible clashes during the implementation. E. g. our implementation does not seem to be involved with any of the syntactic extension anyhow and thus we do not consider these here.

5.1.1 Multi–Parameter Type Classes

The GHC allows to declare multi-parameter type classes with the extension *Multi–Parameter Type Classes*. This is not a standard feature of Haskell2010 and we do not provide support for superclass default instances of such classes.

Note that in the case the superclass default instance is being provided in a class that has more type variables then its superclass it is necessary to address an issue of possibly ambiguous type.

5.1.2 Default Method Signatures

The extension *Default Signatures* allows to specify different signature of default method of the class. The syntax of this extension collides with the syntax of our extension and causes reduce/reduce conflict in the parser. However, this is an implementational detail and does not involve any use of our extension.

5.1.3 Functional Dependencies

The *Functional Dependencies* extension allows programmer to constrain parameters of type classes. This requires the *Multi–*

[2] Currently the GHC data type `ExtensionFlag` that describes available extensions contains 89 constructors.

Parameter Type Classes extension to be active thus same approach as in the case of later extension applies.

5.1.4 Flexible Instances and Undecidable Instances

These two extensions relax constraints on instance context. A context of a superclass default instance is dependent on a class where is this instance declared and is described in the Section 4.1. This context is restricted enough in the terms of Haskell 2010 and further relaxation on constraints does not involve our extension.

5.2 Comparison of Current Solutions to Default Superclass Instances

In this section we provide alternatives to the solutions presented in the Section 2. We discuss merits of our extension compared to the original solution.

5.2.1 Deprecation with Compiler Support

This solution presented in the case of Functor–Applicative–Monad instance is possible to overcome by adding the `Applicative` into the context of the class `Monad` and providing appropriate superclass default instances:

```
{-# LANGUAGE SuperclassDefaultInstances #-}

class Applicative m ⇒ Monad m where
        (>>=)       :: ∀ a b. m a → (a → m b) → m b
        (>>)        :: ∀ a b. m a → m b → m b
        return      :: a → m a
        fail        :: String → m a
        m >> k = m >>= λ _ → k
        fail s = error s

        default instance Fmap m where
                fmap f a = a >>= (λx → return (f x))

        default instance Applicative m where
                pure a = return a
                (<*>) :: f (a → b) → f a → f b
                f <*> a = a >>= (λx → fmap (λg → g x) f)
```

This change in the library is backward compatible thus it can be deployed immediately. The advantage is there is no transitional period with deprecation we discussed in the Section 2.2 and it is not neccessary to manualy change exisiting source code.

5.2.2 Subclass to Superclass Instance

Our solution allows programmer to provide the same functionality as the solution in the Section 2.3. Unlike that solution our approach does not require problematic `UndecidableInstances` extension.

5.3 Applications of Proposed Extension

In this section we given three examples of concrete changes in the hierarchy of standard classes, were already discussed throughout Haskell community, that our extension makes possible.

5.3.1 Bind and Pointed

Edward Kmett has pointed out [13] that although current classes `Functor`, `Applicative`, and `Monad` may be refactored into more convenient structure from categorical point of view it is not to the benefit of programmer without some kind of superclass default instance mechanism. The new structure is shown in Figure 1. Our mechanism allows such refactoring.

```
{-# LANGUAGE SuperclassDefaultInstances #-}

class Functor f where
        fmap :: (a → b) → f a → f b

class Functor f ⇒ Pointed f where
        pure :: a → f a

class Functor f ⇒ Bind f where
        bind :: f a → (a → f b) → f b

class Pointed f ⇒ Applicative f where
        (<*>) :: f a → f (a → b) → f b
        default instance Functor f where
                fmap f a = a <*> pure f

class (Bind m, Applicative m) ⇒  Monad m
        return :: m a
        (>>=) :: m a → (a → m b) → m b

        default instance Bind m where
                bind = (>>=)

        default instance Pointed m where
                pure = return

        default Applicative Bind m where
                mf (<*>) ma = mf >>= λ f → ma >>=
                        λ a → return (f a)
```

User of the library can provide instance of `Monad` with methods `return` and $(>>=)$, which are distributed to superclasses via default superclass instances. Note that it is necessary to use placeholder name `bind` in the `Bind` class in order to allow ordinary instance of `Monad`.

5.3.2 Standard Numeric Classes

There has been discussion on the design of standard numeric classes. According to [7, 22] there are several problems—one of them that standard classes are not finely-grained enough. With our extension it is possible to refactor current structure in a more apt one and maintain backward compatibility.

One of the issues of the critique was the `Num` class. It couples operation for addition and multiplication. It is possible to separate these operations into specific classes:

```
{-# LANGUAGE SuperclassDefaultInstances #-}

    class Additive r where
        add :: r → r → r

    class Multiplicative r where
        mul :: r → r → r

    class (Eq a, Show a) ⇒ Num a where
        (+), (*) :: r → r → r

        default instance Applicative a where
                x `add` y = x (+) y

        default instance Multiplicative a where
                x `mul` y = x (*) y
```

The `Num` class here is simplified for illustrations purposes. Note that this example also demonstrates other problem with numeric classes, it is not obvious whether the operation `(+)` is commutative. On the other hand it is possible to define, e. g., instance of `Additive` for functions of type `Int -> Int` and semantics $(f + g)(x) = f(x) + g(x)$, which is not possible for the class `Num` due to `Eq` and `Show` superclass constraints.

5.3.3 Traversable

The documentation of `Traversable` package [17] currently states properties that instance of the class is expected to satisfy with respect to the classes `Foldable` and `Functor`. However, it is up

to the programmer to ensure this. It is possible with the Superclass Default Instances extension to provide instances satisfying these rules automatically and thus avoid possible inconsistencies:

```
{-# LANGUAGE SuperclassDefaultInstance #-}
newtype Id a = Id { getId :: a }
newtype Const a = Const { getConst :: a }

instance Functor Id where
        fmap f (Id x) = Id (f x)

instance Traversable (Const m) where
        traverse _ (Const m) = pure (Const m)

class (Functor t, Foldable t) ⇒ Traversable t where
        ...

        default instance Functor t where
                fmap f = getId ∘ traverse (Id ∘ f)

        default instance Foldable t where
                foldMap = getConst ∘ traverse (Const ∘ f)
```

This example is incomplete and serves only the illustrative purposes. The full example is provided with the enclosed implementation.

6. Implementation

This section briefly describes an implementation details of the language proposal we introduced in the Section 4. We have selected the GHC as a compiler into which we incorporate our extension. We consider the GHC [9] documentation on compiler internals and compiler development superior to other compilers (e. g. Utrecht Haskell Compiler [6]).

6.1 Compiler Architecture

Marlow and Peyton–Jones presented a architecture of GHC in [15]. They state modularity and the openness to the research and compiler extension to be one of the project goals. The modularity allows us to describe both the architecture and consecutively the changes to the compiler in several detached steps.

In general the GHC project involves more than just the compiler itself. Precisely it contains the compiler, the basic libraries that the compiler depends upon, and The Runtime System (RTS) that handles running the compiled code. We do not need to take care of the architecture of neither libraries nor the RTS. The libraries contain general data structures, e. g. `Data.Map`, and although we make use of them, we do not need to make any changes.

The compiler processes source program into *Haskell core*, a variant of system F called FC [20]. Any higher syntactic construct is translated into the core at first and the code generation follows after this process. Thus we do not need to take the RTS into account when describing our implementation as no adjustments to it take place. The compiler is further divided into three parts:

- The *compilation manager* handles compilation of multiple source files. Its task is to decide which files need to be recompiled because some of the dependencies have changed since the last compilation.
- The *Haskell compiler (Hsc)* that handles compilation of single file.
- The *pipeline* composes any external programs (e. g. C preprocessor) with the Hsc.

Only the Hsc is of our concern as we do not change behavior of multiple file compilation nor manipulate with any external programs. Compiling a Haskell source file proceeds sequentially in several phases. The structure of the phases that are important with respect to our implementation is illustrated in the Figure 6.1

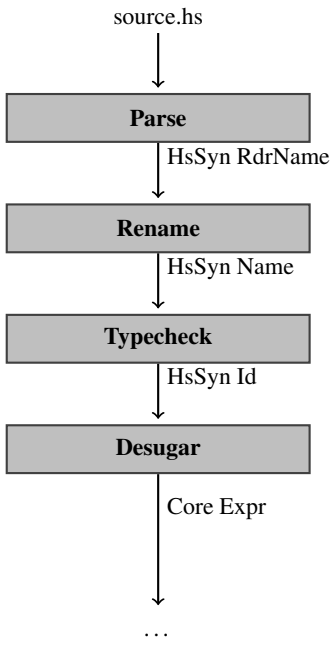

Figure 5. The compilation phases

In Parser phase a source file is converted into abstract syntax. A lexical analyser and parser are involved. The abstract syntax data type is the `HsSyn t` for some type of identifier `t`. The parses produces the original string names `RdrName` from source code. Hence, the type of abstract syntax is `HsSyn RdrName`. This part of compiler uses the Alex library [1] for generating lexical analyser and the Happy library [11] as a parser generator.

Renamer compilation phase converts all identifiers into the fully qualified names. The identifier types are resolved from `RdrName` to references to particular entity `Name`. Therefore, the resulting abstract syntax has type `HsSyn Name`. Type checker verifies that the Haskell program is type-correct. The type checker resolves the types of identifiers resulting in conversion of `Name` type in abstract syntax to `Id` type.

Following the type checker desugaring takes place. Desugaring translates all higher language constructs into basic core language. Then the core code is simplified through optimization, e. g., dead code elimination and case expression reduction. After these phases the target code generation follows. Particular behavior depends on compiler settings–either native machine code, LLVM code, or C code may be produced. In this process GHC also generates interface files in order to support separate compilation.

6.2 Changes to the Compiler

This section briefly documents changes to the compiler and design of our implementation. In the description of the changes we follow the sequential architecture of the compiler and discuss each part of the compiler respective to the compilation phase separately. We note only the most important changes. In particular, we show changes to data structures involved in compilation process.

6.2.1 Extension Flag

Before proceeding with the implementation have we added the *Extension Flag* and registered the flag with a command line option.

This allows programmer to enable the extension from source code and from the command line respectively. The appropriate data types are located in the file `main/DynFlags.hs`.

```
data ExtensionFlag
       = Opt_Cpp
       | Opt_OverlappingInstances
       ...
       -- | Our extension flag
       | Opt_SuperclassDefaultInstances
deriving (Eq, Enum, Show)

xFlags :: [FlagSpec ExtensionFlag]
xFlags = [
  ( "CPP",
      Opt_Cpp, nop ),
  ...
  -- Our extension
  ( "SuperclassDefaultInstances",
      Opt_SuperclassDefaultInstances, nop )
]
```

This allows us to recognise whether the extension is enabled in the compilation context.

6.2.2 Parser

Our extension dost not add nor any keywords nor new lexical forms to the language. Thus we do not alter the lexer. We incorporate changes into the language grammar to the parser definition, which is implemented in `parser/Parser.y.pp`. Particularly we modify the class declaration rule with our default instance branch

```
decl_cls  : at_decl_cls { LL (unitOL $1) }
        | decl          { $1 }
        ...
        | 'default' dinst_decl {%
                hintSuperclassDefaultInstances (getLoc $1)
                return ...
        }
```

and introduced default instance rule:

```
-- Default instances
dinst_decl : 'instance' inst_type where_inst { ... }
```

Note that the function `hintSuperclassDefaultInstances` is a new helper that checks the extension flag and causes appropriate compilation error if the flag is not present. For technical reasons we require user to explicitly state the context of default instance.

6.2.3 Renamer

We have introduced the method `rnClsDefInstDecl` in `rename/RnSource.lhs`. The method handles renaming in the case of superclass default instances. It is modeled after the appropriate method that handles ordinary instance renaming.

6.2.4 Type Checker

We have modified basic data types representing declarations in the file `hsSyn/HsDecls.lhs`. In particular, we have modified the `ClassDecl` constructor of `TyCllDecl` data type, which represents type or class declaration, in such manner it now carries the information about superclass default instances. We have also introduced a data type that represents superclass default instance:

```
data ClsDefInstDecl name = ClsDefInstDecl {
    -- Context ⇒ Class Instance-type
    cdid_poly_ty :: LHsType name
    , cdid_binds :: LHsBinds name
    , cdid_sigs  :: [LSig name] --user supplied
    -- ^ type family instances
    , cdid_tyfam_insts :: [LTyFamInstDecl name]
    -- ^ data family instances
    , cdid_datafam_insts :: [LDataFamInstDecl name]
    , cdid_overlap_mode :: Maybe OverlapMode
    }
deriving (Data, Typeable)
```

We have also enriched global environment `TcGblEnv` with default instance environment `tcg_dinst_env` and list of default instances `tcg_dinsts`:

```
data TcGblEnv
  = TcGblEnv {
    tcg_inst_env     :: InstEnv,
      -- ^ Instance envt for all /home-package/ modules;
      -- Includes the dfuns in tcg_insts
    tcg_dinst_env    :: InstEnv,
      -- ^ Ditto for default instances
    ...
    tcg_insts     :: [ClsInst],  -- Instances
    tcg_dinsts    :: [ClsInst],  -- Default Instances
    ...
}
```

This allows us to collect default instances separately from ordinary instances.

6.3 Desugaring

We use existing machinery that handles desugaring of ordinary instances also for desugaring of superclass default instances.

7. Further work

The future work related to this thesis embodies mostly in two directions. Having this implementation it is important to asses it on real world examples whether the proposal is fit to be accepted into GHC and which changes can be made in the standard library with the use of our extensions.

Beside the practical work there is also the other direction. The more theoretical direction of the work consists of possible opt-out mechanism for default instances. We do not discuss these in out work although they are considered in the previous proposals. The other proposal – class aliases – could be also used to a great benefit with superclass defaults. However, following discussion in this paper, proper specification behavior of such aliases is necessary.

There are also some other language proposals that are loosely related to the default instances, e. g. quantified contexts [19]. Identification of these proposals and their relation to superclass defaults seems to us as a solid base for further work on the language extension that the language and its users could benefit from.

Conclusion

We have described a maintainability problem with Haskell type classes that occurs in practice. We have summarized previous attempts to solve this problem and analysed different features of this approaches. Based on this analysis we have derived a proposal for a language extension – the Default Superclass Instances Extension – that solves the problem. The proposal allows programmer to declare default instance of a superclass within a class declaration. We have demonstrated on several examples how the superclass default instances solve the problem under our consideration. We have discussed relations of our extension to existing extensions and provided sample solutions to discussed problem. We have also implemented the extension and we have described the main design choices of this implementation.

The accompanying implementation of our proposal shows that our proposal is solid and can be used in practice. Listed sample solutions only cover a small set of problems whether the real application of our work could be much wider. Introduction of these

language extensions into mainstream compiler would allow on one hand to correct some existing problems in class dependencies and on the other hand the authors of libraries would have more freedom in the design of such libraries knowing that any design choice can be corrected in future.

References

[1] *Alex: A lexical analyser generator for Haskell*. Tech. rep. http://www.haskell.org/alex/.

[2] *Applicative/Monad proposal related warnings (AMP phase 1)*. Online. Feb. 2015. URL: https://ghc.haskell.org/trac/ghc/ticket/8004.

[3] Richard Bird et al. "Understanding Idiomatic Traversals Backwards and Forwards". In: *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*. Haskell '13. New York, NY, USA: ACM, 2013, pp. 25–36.

[4] *Class system extension proposal*. Online. Mar. 2012. URL: http://www.haskell.org/haskellwiki/index.php?title=Class_system_extension_proposal&oldid=44718.

[5] *Default superclass instances*. Online. July 2014. URL: http://ghc.haskell.org/trac/ghc/wiki/Default/SuperclassInstances?version=30.

[6] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. "The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity." In: *IFL*. Ed. by Olaf Chitil, Zoltn Horvth, and Viktria Zsk. Vol. 5083. Lecture Notes in Computer Science. Springer, 2007, pp. 57–74.

[7] Mikael Johansson Dylan Thurston Henning Thielemann. *The numeric-prelude package*. July 2014. URL: http://hackage.haskell.org/package/numeric-prelude-0.4.1.

[8] *Functor–Applicative–Monad Proposal*. Online. July 2014. URL: http://www.haskell.org/haskellwiki/index.php?title=Functor-Applicative-Monad_Proposal&oldid=58553.

[9] *GHC Documentation*. Tech. rep. July 2014. URL: http://www.haskell.org/ghc/docs/7.8.3/html/.

[10] Cordelia Hall et al. "Type Classes In Haskell". In: *ACM Transactions on Programming Languages and Systems* 18 (1996), pp. 241–256.

[11] *Happy: The Parser Generator for Haskell*. Tech. rep. http://www.haskell.org/happy/.

[12] *Implement Functor ⇒ Applicative ⇒ Monad Hierarchy (aka AMP phase 3)*. Online. Feb. 2015. URL: https://ghc.haskell.org/trac/ghc/ticket/4834.

[13] Edward Kmett. *Lens based classy prelude*. Online. Sept. 2013.

[14] Simon Marlow. *Haskell 2010 Language Report*. Tech. rep. June 2010.

[15] Simon Marlow and Simon Peyton-Jones. "The Glasgow Haskell Compiler". In: *The Archicture of Open Source Applications, Volume II: Structure, Scale, and a Few More Fearless Hacks*. Ed. by Greg Wilson and Amy Brown. Vol. ii. Self published, Apr. 2012. Chap. 3.

[16] Conor McBride. *the Strathclyde Haskell Enhancement*. Online. July 2014. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/.

[17] Conor McBride and Ross Paterson. *Data.Traversable*. Online. 2005. URL: https://hackage.haskell.org/package/base-4.7.0.0/docs/Data-Traversable.html.

[18] John Meacham. *Class Aliases*. Online, July 2014. URL: http://repetae.net/recent/out/classalias.html.

[19] *Quantified contexts*. Online. May 2010. URL: http://www.haskell.org/haskellwiki/index.php?title=Quantified_contexts&oldid=34638.

[20] Martin Sulzmann et al. "System F with type equality coercions." In: *TLDI*. Ed. by Franois Pottier and George C. Necula. ACM, 2007, pp. 53–66.

[21] *Superclass defaults*. Online. Dec. 2007. URL: http://www.haskell.org/haskellwiki/index.php?title=Superclass_defaults&oldid=17441.

[22] Henning Thielemann. "How to Refine Polynomial Functions". In: *IJWMIP* 10.3 (2012).

[23] John Wiegley. *Proposal: Add Data.Semigroup to base, as a superclass of Monoid*. June 2013. URL: https://mail.haskell.org/pipermail/libraries/2013-June/020188.html.