Controlling the scope of instances in Haskell

Marco Gontijo Carlos Camarão

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

marcot@ufmg.br camarao@dcc.ufmg.br

Abstract. The Haskell module system aims for simplicity and has a notable advantage of being easy to learn and use. However, type class instances in Haskell are always exported and imported between modules. This breaches uniformity and simplicity of the module system and introduces practical problems. Firstly, it is not possible to define two distinct instances of the same type class for the same type in a program. Secondly, instances created in different modules can conflict with each other, and can make it impossible to import two modules that contain instance definitions even if the instances are not used in the importing module. In this paper we present and discuss a solution to these problems, that simply allows importation and exportation of instances between modules. We also show how a formal specification of the module system must be adapted to handle instances and to include our proposal.

1 Introduction

Modern programming languages promote code reuse by supporting polymorphism, which allows the same code to be used with distinct data types. There are different approaches to polymorphism, one of them being ad-hoc, or constrained, polymorphism [12], which support code that use overloaded names (or symbols) and reuse of such code for all data types for which a definition of the overloaded names have been given. Type classes are a language mechanism that was introduced in the programming language Haskell for supporting ad-hoc polymorphism [4]. A type class specifies a set of overloaded names together with type annotations for them. An implementation of a type class for a data type, called an instance of the type class, provides definitions for all overloaded names of that type class. In this paper we propose a change to the module system of Haskell, a language that is nowadays used in academic research, specially to study and experiment with topics related to type systems and type inference, and is also being used in commercial applications[1]. Our proposal is related to the way instance definitions are handled in Haskell's module system.

A module system of a programming language is intended to provide support for modular construction of software systems. In some languages the module system provides a type-safe abstraction mechanism, where module definitions

[1] http://industry.haskell.org/

can be parameterized so that modules can be instantiated by means of different kinds of entities. This is the case for example of Standard ML [8] and Scala [9]. A module system can also merely allow a program to be divided into parts that can be compiled separately. In some other languages, the module system provides a mechanism to control the visibility of globally defined names, either to hide implementation-specific details or to access parts that would otherwise be out of scope. This is the case for example of Haskell [7, chapter 5].

The Haskell module system aims for simplicity[2] and has a notable advantage of being easy to learn and use. However, this simplicity is partly hindered by the special treatment given to the scope of instances. As defined in the Modules chapter of the Haskell 2010 Report [7, section 5.4], a type class "instance declaration is in scope if and only if a chain of `import` declarations leads to the module containing the instance declaration".

Because of this, it is not possible to define two instances of the same type class for the same type in a set of modules that are related by an import or export chain. The modules that compose a Haskell program are always in the same import or export chain, because the main module imports all the modules used in the program, directly or indirectly. So, it is not possible to define two instances of the same type class for the same type in the same program. This is a serious restriction. The aim is, as in all type system restrictions, to prevent the programmer from making mistakes. However, even though this design decision protects the programmer from incurring in some mistakes, it can also disallow reasonable and correct code. Furthermore, a lot of instances generally become part of the scope of modules without ever being used. This puts a burden on compiler writers, which have to consider smart ways of controlling the size of the scope of modules.

In this paper we propose an extension to the Haskell language, which allows programmers to control when to export and import instances. This makes it possible to create instances local to a module or visible only in a subset of modules of a program, and removes problems brought by importation of modules that contain definitions of instances for the same type, as described in detail in section 2 (subsection 2.2). This section also illustrates how the abscence of control of the visibility of instances makes it hard or impossible to use instances for a certain type with a special purpose (subsection 2.1). In the third section we present our proposal, with two possible alternatives, also discussing its implementation, and a complementary proposal for giving names to instances. This section includes a discussion about problems that can occur by the adoption of our proposal, and possible solutions to them. The fourth section describes one way of extending a published formalization of Haskell's module system [1] in order to handle instances, both with and without our proposal. The fifth section describes related work and the final section concludes the paper.

[2] As stated by Simon Peyton-Jones in his interview entitled "The A-Z of Programming Languages: Haskell", available at `http://www.computerworld.com.au/article/261007/a-z_programming_languages_haskell/?fp=16\&fpid=1` .

Fig. 1. Example of the usage of `newtype` to create a new instance.

```
import Data.List
newtype IChar = IChar Char
unbox :: IChar -> Char
unbox (IChar c) = c
instance Eq IChar where
  (IChar c1) == (IChar c2) = iEq c1 c2
instance Ord IChar where
  compare (IChar c1) (IChar c2) = iCmp c1 c2
iSort :: [String] -> [String]
iSort = map (map unbox) . sort . map (map IChar)
```

2 Background

2.1 Defining special purpose instances

Since it is impossible to define more than one instance for a given type, the programmer can not, for example, sort the same type of data by using two different techniques and by applying a function `sort`. As a more specific example, a programmer can not use case-sensitive ordering to sort a list of strings in a part of the program and case-insensitive ordering in another.

A general way to work around this problem is to create a new data type encapsulation, using `newtype`, and define a different instance for it. The example in Figure 1 illustrates this solution. This works, but it is verbose and not efficient. In other words, it is "too clunky"[3]. It is a simple solution that can be considered good enough for this problem, but it does not address the problem of the pollution of the global scope.

A less verbose solution exists, with the definition and use of functions that include additional parameters instead of methods of type classes. For example, module `Data.List` defines function `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`, which sorts the list passed as the second parameter using the comparison function given by the first parameter. This is a simple and useful solution to this specific problem, but it does not scale well. To apply the same idea generally, for all functions that use a type class method a similar function having an additional parameter used instead of the type class method would be necessary. This is not a reasonable idea because it would add parameters in a lot of cases, making the code more complicated. Also, it goes against the idea of making code simpler and more reusable by means of overloading.

2.2 Orphan instances

The global visibility of type class instances create so-called *orphan instances*. Orphan instances are instances defined in a module that contain neither the

[3] In Lennart Augustsson's words. `http://lukepalmer.wordpress.com/2009/01/25/a-world-without-orphans/#comment-609` .

Fig. 2. Module T.

```
module T where
class T a where
  t :: a
```

Fig. 3. Module D.

```
module D where
data D = D
```

Fig. 4. Module I1.

```
module I1 where
import T
import D
instance T D where
  t = undefined
i1 :: a
i1 = undefined
```

Fig. 5. Module I2.

```
module I2 where
import T
import D
instance T D where
  t = undefined
i2 :: a
i2 = undefined
```

Fig. 6. Main module of the example of orphan instances.

```
import I1
import I2
f :: a -> a -> a
f = undefined
g :: a
g = f i1 i2
```

definition of the data type nor the definition of the type class. When an instance is defined in a module where the data type or the type class is defined, it is guaranteed that there will not exist more than one instance for each type class and data type. Orphan instances are, thus, important because, considering the module system, they are the mechanism that enable the creation of distinct instances of a type class for the same data type.

They are specially troublesome when a module defines other functions that are not related with the instance. For example, if we have a module `T` (Figure 2) that defines a type class `T`, a module `D` (Figure 3) that defines a data type `D`, and two modules `I1` (Figure 4) and `I2` (Figure 5) that define instances of `T` for `D`, we would not be able to import both `I1` and `I2` in the same module.

In the example we are more interested in types and visibility control by the module system than in the body of the presented functions. Therefore, we are using function `undefined`, but the problem remains the same if there was a relevant function body.

Instances defined in `I1` and `I2` are orphan instances. The problem gets worse when there is a need to use, in the same module, functions that are not related to instances, like `i1` and `i2`. It is not possible to use `i1` and `i2` on the same program without modifying `I1` or `I2`. Even if `i1` and `i2` are used in different modules, the main module will have to import both of them or a module which imports them. Modifying `I1` or `I2` is not always possible in practice because they may be part of a third-party library.

It is worth noticing that these are not only potential problems. They happen in real world uses of the language. For example, the Monad instance of Either is defined in both packages `mtl` and `transformers`[4]. There are examples where

[4] This example is on the wiki page at `http://www.haskell.org/haskellwiki/Orphan_instance` .

orphan instances would be desirable, involving pretty printing and JSON[5]. Also, a situation has been reported where instances created with Template Haskell could not be defined in the same module of the data type or type class[6].

3 Solution

We propose that instances should be exportable and importable in the module system. It is a natural, obvious proposal that has already been mentioned[7], but this work provides a detailed description and discussion, including required changes in the language definition.

The proposal solves the mentioned problem of the existence of orphan instances. The fact that a module defines an instance without defining the related data type or type class does not cause any bad consequence anymore: the programmer can choose which instance to use by importing one module instead of another, and it can still use functions defined in both modules, by hiding instances in an import clause. The `sortBy` problem is also solved, because programmers can change the instance of a type class for a data type in the context of a module, making it possible to call `sort` with the semantics of the instance defined in this module.

We examine two alternative syntaxes for the new language feature: a backwards compatible one, referred to as **intermediate** — but not very uniform — and a backwards incompatible one, called **final**, which is more uniform.

Before being adopted in the language, these language extensions should preferably be enabled by compilers by the use of a compilation flag. There should exist then a different flag for each of the different syntaxes.

In both cases, the `export` and `import` terms used in The Haskell 2010 Report [7, sections 5.2 and 5.3] are changed to have a new syntax element, which is equal to the header of an instance declaration [7, section 4.3.2]: `instance [scontext =>] qtycls`. This identifies wether an instance should be exported, imported or hidden. The new declarations are given in Figures 7 and 8.

3.1 The final alternative

In the final alternative, instances are imported and exported in the same way as other definitions in Haskell. There are five distinct cases of import clauses affected by the proposal, presented below by considering canonical cases of import clauses applied to module `I1` presented previously, as done in [7, section 5.3.4]:

1. `import I1` imports everything in module `I1`, including instances, as occurs in Haskell currently;

[5] This example was presented by Lennart Augustsson in `http://lukepalmer.wordpress.com/2009/01/25/a-world-without-orphans/#comment-601` .

[6] Johan Tibell gives a detailed description of the situation in an e-mail at `http://www.haskell.org/pipermail/glasgow-haskell-users/2010-August/019052.html` .

[7] By Yitzchak Gale on Stack Overflow `http://stackoverflow.com/questions/3079537/orphaned-instances-in-haskell/3079748#3079748` .

Fig. 7. New syntax for the export clause.

export → qvar
  | qtycons [(..)|(cname$_1$, ..., cname$_n$)]
      ($n \geq 0$)
  | qtycls [(..)|(var$_1$, ..., var$_n$)]
      ($n \geq 0$)
  | `module` modid
  | `instance` [scontext =>] qtycls

Fig. 8. New syntax for the import clause.

import → var
  | tycon [(..)|(cname$_1$, ..., cname$_n$)]
      ($n \geq 0$)
  | tycls [(..)|(var$_1$, ..., var$_n$)]
      ($n \geq 0$)
  | `instance` [scontext =>] qtycls

2. `import I1 ()` imports nothing, as occurs as if this line was commented or not present in the source code;
3. `import I1 (instance T D)` imports only the instance, which would be the same as `import I1 ()` in Haskell 2010;
4. `import I1 hiding (instance T D)` imports everything but the instance;
5. `import I1 (i1)` imports only `i1`, and not the instance.

The only instance defined in `I1` is `instance T D`. If there were other instances to be imported, they should be also included where `instance T D` is listed.

Similarly, there are four cases of export clauses affected by the proposal:

6. `module I1 where` exports everything in `I1`, including the instance, as occurs in Haskell currently;
7. `module I1 () where` exports nothing, not even the instance;
8. `module I1 (instance T D) where` exports only the instance, such as `module I1 () where` in Haskell 2010;
9. `module I1 (i1) where` exports only `i1`, and not the instance.

This syntax is not backwards compatible because the behavior of a program that contains a line like (2), (5), (7) or (9) is correct in Haskell 2010, but has a different meaning than the one we are proposing. In Haskell 2010, the instance is imported/exported in all cases, and in our proposal, the instance is not imported/exported in all cases. We propose that this language extension should be incorporated in the language in a second step, with an intermediate step that is the adoption of the intermediate alternative.

3.2 The intermediate alternative

The intermediate alternative is mostly similar to the final alternative, but exceptions are made to make it backwards compatible. In (2), (5), (7) and (9) the instances are imported/exported, and the only way to avoid the instance from being imported is by using the keyword `hiding` in the import list. There is no way to avoid it from being exported. In the intermediate alternative, (8) is valid and has the same effect as (7).

The semantics of the intermediate alternative can be expressed using the syntax of the final alternative. The interpretation of the examples that have their meanings changed are rewritten on Figure 9. As the intermediate alternative has

Fig. 9. The semantics translation from the intermediate syntax to the final.

	Intermediate (or Haskell 2010)	Final
2	`import I1 ()`	`import I1 (instance T D)`
5	`import I1 (i1)`	`import I1 (i1, instance T D)`
7	`module I1 () where`	`module I1 (instance T D) where`
9	`module I1 (i1) where`	`module I1 (i1, instance T D) where`

a syntax that is backwards compatible with Haskell 2010, Figure 9 also shows how Haskell 2010 constructs are mapped to the syntax of the final alternative.

The intermediate alternative has the same advantages of the final alternative, but it is less uniform and should be used temporarily while programs are adapted to use the syntax of the final alternative. During this period, using constructions (2), (5), (7) and (9) should be considered as a bad programming practice. These should be gradually replaced by their final version, as shown in Figure 9. The final version is also a valid intermediate syntax program, with the same meaning.

After this period, when the syntax of the final alternative becomes used, the use of these constructions — that is, (2), (5), (7) and (9) — should be acceptable, but they will have the semantics defined here, and not the old semantics.

New languages claims to justify their existence fall under three categories [6, p. 1]: "novel features, incremental improvement on existing features, and desirable language properties". This paper presents a language extension, which also needs a justification. Our proposal as a whole can be seen as incremental improvement on existing features, because it is not creating something new, but it is improving the use of something that already exists. The difference between the intermediate and the final variations brings desirable language properties, which is uniform behavior for similar constructs.

3.3 Instance names

A complementary syntax that could be added as an extension, and enabled by a compiler using yet another compilation flag, is the attribution of names to instances. The motivation for this is that sometimes instance contexts and types that identify instances can be quite long and complex. For example, `instance (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h, Eq i, Eq j, Eq k, Eq l, Eq m, Eq n, Eq o) => Eq (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o)` is defined in the Haskell Prelude. It would be better to create a name for this instance, like EqTuple15, and use this name in import and export lists.

This, as the rest of the proposal, would syntactically affect only the module system. The programmer will be able to create a synonym to refer to the instance in export and export lists. The idea of creating a synonym is similar to the `type` construction in Haskell.

Naming of instances can be done using a top-level declaration like in, for example, `inst Inst1 = instance T D`. After an instance synonym is declared, it would be possible to use the introduced name on import and export lists. For instance: `import I1 hiding (Inst1)`.

Fig. 10. Second version of module I1, using the proposed extension.

```
module I1 where
import T
import D
inst Inst1 = instance T D
instance T D where
  t = undefined
-- i1 :: D
i1 = t
```

Fig. 11. Second version of the main module, using the proposed extension.

```
import I1 hiding (Inst1)
import I2
f :: D -> b -> b
f = undefined
g :: a
g = f i1 i2
```

Although it has a similar name, the Named Instances proposal [5] is very different from ours, because it requires more significant changes to the language. More details about how our work is related to others is present on Section 5.

3.4 Instance scope

Although the control of the visibility of instances allows control of which entities are necessary and should actually be in the scope of modules, there are subtle and somewhat unfortunate consequences of such control. The most notable one is that a type annotation may cause the semantics of the annotated construct to be changed.

To see this, consider the example in Figure 10, and two cases. In the first, there is no type annotation of the type of function `i1`, or there is an annotation, like `i1 :: T a => a`, that does not instantiate the constraint on `T`. In the other case, the type of `i1` is annotated so as to instantiate the constraint on `T`, as for example `i1 :: D`.

If the main module (Figure 11) did not import module `I2`, it would not be able to instantiate function `i1` to `D`. In the example presented, it will instantiate the function to `D`, but using the instance defined in `I2`. Therefore, the writer of module `I1` should notice that the instance defined there will not necessarily be visible in the imported module and, when there is an instance visible, it will not necessarily be the one defined in module `I1`.

Also, the programmer should be aware that if the type annotation is included, by uncommenting the line in module `I1`, the instance defined in module `I1` will be used, even though it is not visible in module main. As already stated, if the line is commented, the instance defined in `I2` will be used.

3.5 Implementation

Usually, a compiler keeps a list of available instances while building a module. This list is used to check if an instance is available when inferring and checking types, and to choose which instance to use when generating code. Currently, instance visibility can not be controlled, so instances are only included in this list, and there is no need for compilers to remove any element of this list. The

Fig. 12. Module Definition, used in the example of unexpected behavior that arises from misuse of local instances.

```
module Definition where
import Data.Set
s :: Set Char
s = insert 'a' $ insert 'B'
  $ empty
```

Fig. 13. Main module of the example of unexpected behavior that arises from misuse of local instances.

```
import Definition hiding
  (instance Ord Char)
import Prelude hiding
  (instance Ord Char)
instance Ord Char where
  compare = iCmp
m :: Bool
m = member 'a' s
```

implementation of our proposal will require removing elements from this list while importing and exporting definitions from a module.

Our proposal aims to be simple and require as few changes to the language as possible. This is noticed when the implementation details are made clear: it is only a matter of filtering imported or exported instances when requested.

3.6 Problems and Solutions

Like most changes to an established language, this proposal has its pros and cons. Considering that "a new language feature is only justifiable if it results in a simplification or unification of the original language design, or if the extra expressiveness is truly useful in practice" [10, p. 1], we judge that this language feature is justifiable because the extra expressiveness added to Haskell is truly useful in practice. The main force that pushes research in this field is the desire to have more well typed programs [11, p. 3], and this is our motivation.

On the other hand, there are reasons why this proposal was not included in the language in the first place. It may be argued that changing the definition of an instance of a class to a type in a program makes it harder to understand what the code means. This is only a problem if the changes made to the definitions are not intuitive in the program context, and this is not a problem of the language extension per se, but of a possible use of it. In Haskell, it is already possible to break intuitivity with expressions like `let 1 + 1 = 3 in 1 + 1`, which overloads a function in local scope, without properly changing the related type class or its instances. So, this is not going to be the only case in the language where basic constructions can have their meaning changed.

Changes to instance definitions can cause potentially unexpected things to happen. Consider the following example. Suppose that a value of type `Set` is internally represented by an ordered structure of its elements, and that is why common operations, like insert, requires the type to be an instance of `Ord`. If a value of type `Set Char` is defined in a module where the visible instance of `Ord Char` is the default, and then used in a module where a case-insensitive instance is visible, the search operation can give perhaps unexpected results.

In module `Definition` (Figure 12) `'a'` will be inserted after `'B'`, since in case-sensitive order it comes later. Suppose `iCmp` is the comparison function for case-insensitive Char. The call of `member` on the main module (Figure 13) will search for `'a'` before `'B'`, because that is the case-insensitive order, and it will not find it, returning `False`. This is arguably not a good thing, but it is caused by a misuse of a feature. Dealing with it requires programmers to be careful when using different instances of a type class for the same type in programs.

Another issue is related to the fact that the semantics of a function may change because of the inclusion or not of a type signature.[8] Although this is in general undesirable, in this case, when a type is annotated with a less general type, an instance is being chosen. The instance to be used should be the one available in the module where it was chosen, and not in the module where the exported function is used. In the example with the module `I1`, if the type of `i1` is annotated as `D`, the choice of which function is used is made in module `I1`, and thus the instance defined in `I1` must surely be the instance used.

A Haskell module exports functions with defined types, and a type annotation can change a defined type. If a module exports a function with a type such as, for example, `Num a => a -> a`, the insertion of a type annotation can change this type, for example to `Int -> Int`. A module that imports this function, and uses it with type `Integer -> Integer` will not compile, even if the function definition remains the same. Thus, a type annotation included in a top level declaration can change the interface of a module, and it is reasonable that some programs will then stop working. When the interface of a module changes, because of a change in the type of an exported function, it is reasonable that the semantics of the exported function can change.

Our proposal makes it possible for a change in type annotations to cause semantic changes, but only between modules and not inside a module. Such a semantic change can occur only when the interface of a module changes, by a change in the type of an exported function. In the example, function `i1` with type annotation `D` is not, in any way, related to type class `T`, and should thus not be affected by instances declared in the importing module. On the other hand, if no type is annotated, or a type that has a constraint on `T` is annotated, function `i1` will be related to the type class, and its use can thus be affected by the definition or existence of instances of this type class. Notice that there exist already other examples of cases of type annotations affecting the semantics of Haskell programs, related to the use of defaulting rules[9] and an "a *really* amazing

[8] Simon Peyton-Jones states that type annotations should not change the result of a function in this e-mail: `http://www.haskell.org/pipermail/haskell/2001-May/007111.html` .

[9] Described in e-mails `http://www.haskell.org/pipermail/haskell/2001-May/007113.html` , `http://www.haskell.org/pipermail/haskell/2001-May/007118.html` and `http://www.haskell.org/pipermail/haskell/2001-May/007117.html` .

Fig. 14. Auxiliary functions for filtering instances in the module system.

```
isInst :: Entity -> Bool
isInst (Entity { name = n }) = head (words n) == "instance"
isInst _ = False
instances :: (Ord a) => Rel a Entity -> Rel a Entity
instances = restrictRgn isInst
```

example"[10] using polymorphic recursion[11]. We believe that the advantages of our proposal outweigh disadvantages related to these issues.

4 Extending Haskell's Module System Formal specification

The module system of Haskell 98 has been formally specified [1] without dealing with type class instances. This section presents an extension of this formalization for dealing with type class instances, including the changes needed in [1] in order to cope with both the intermediate and final alternatives of our proposal. The paper in which the formalization is made does not provide the complete code of the formalization, but the code is available on the web[12].

The code models `Name` as a wrapper around a `String`, and it is stated in the paper that type class instances were not considered because it is not possible to refer to them by a name [1, section 3.1]. We propose that names of instances be written as they occur in export and import clauses (as presented in Figures 7 and 8). By doing this, there is no need to change data type `Name`, nor data type `Entity` used for describing exported and imported entities.

For the Instance Names extension, presented in Section 3.3, instance names can also be used to refer to an instance. In this case, the name mentioned in the `Entity` data type must be the real name of the instance, and not the synonym. Otherwise, it will not be possible to tell if the name refers to an instance or not: the auxiliary function `isInst`, defined in Figure 14, is used to distinguish type class instances from other entities. Funcion `isInst` is used in the same manner as function `isCon`, defined in the paper [1, section 3.1]. Another auxiliary function that should be defined is a filter for type class instances, called, say, `instances` (see Figure 14), to be used for the changes introduced in our extension of the formalization.

[10] As mentioned by Simon Peyton-Jones in `http://www.haskell.org/pipermail/haskell/2001-May/007133.html` .

[11] Described by Lennart Augustsson in `http://www.haskell.org/pipermail/haskell/2001-May/007122.html` .

[12] `http://yav.purely-functional.net/publications/modules98-src-21-Nov-2005.tar.gz`.

4.1 Haskell and the intermediate alternative

Our proposal can be applied to both Haskell 98 or Haskell 2010, since the language changes from Haskell 98 to Haskell 2010 do not affect the proposal. The changes needed to be done in the formalization of the module system for including the way Haskell deals with type class instances and the way our intermediate proposal deals with it are the same. The difference is that our proposal provides some syntatic constructs which are not available in Haskell. From the perspective of the module system specification, this will mean that some possibilities, like hiding an instance, are not going to happen, but having the code for it available will not interfere with the result. Because of this, in this subsection we present the changes needed for both Haskell and our intermediate proposal.

Only two things need to be changed in the specification: the way exported and imported entities are obtained. In the case of exported entities, function `exports` [1, section 5.2] needs to be changed. The old version of the function is presented in Figure 15 and the new version in Figure 16. The difference between them is just that, when a export list is available (the `Just es` case) the instances are exported with what is on the export list. The instances, then, are always exported, as defined in Haskell 2010 report [7, section 5.4].

Fig. 15. Function `exports` as in [1, section 5.2].

```
exports :: Module ->
  Rel QName Entity ->
  Rel Name Entity
exports mod inscp =
  case modExpList mod of
    Nothing -> modDefines mod
    Just es -> getQualified
        `mapDom` unionRels exps
      where
      exps = mExpListEntry inscp
        `map` es
```

Fig. 16. New function `exports`.

```
exports :: Module ->
  Rel QName Entity ->
  Rel Name Entity
exports mod inscp =
  case modExpList mod of
    Nothing -> modDefines mod
    Just es -> unionRels
        [getQualified
          `mapDom` unionRels exps,
        instances
          $ modDefines mod_]
      where
      exps = mExpListEntry inscp
        `map` es
```

The other change needed, which is related to imported entities, is on function `mImp`. The change deals with a function defined in the `where` clause of function `incoming`. The old and new versions of function incoming are presented respectively in Figures 17 and 18. Similarly to the change in the `exports` function, this change includes instances in entities that are going to be imported even if they are not in the import list.

Notice that, in the case of a hiding import such that an instance is on the hiding list, in the intermediate alternative the instance will not be imported, as expected, because instances are only being added in the case where they are

not a hiding import. Also, if the instance is not on the hiding list, it will be imported, because it is included in `exps`.

4.2 The final alternative

To specify the final alternative, the consideration about how to use the instances as names is still valid, in order to allow the system to recognize instances, but the rest of the specification must be kept in the same way as it is, that is, without the changes proposed in the last subsection. This happens because our proposal makes instances be treatable in the same fashion as other Haskell entities, so that the specification that worked for them works also for instances.

5 Related work

The issues that Named instances [5] solves intersect with the issues discussed in this paper. It creates a new name for each instance, which should be informed on function call. This implies big changes on the language, including "how much context reduction should be done before generalization" [10, p. 8]. Our proposal is simpler, since it requires fewer changes in the language and is, therefore, more likely to be included and internalized by Haskell programmers.

Named instances provide more expressivity than our proposal, because it allows any two different instances of the same type class for the same data type to be used in the same module. In our proposal, two different instances of the same type class for the same data type can only be used in two different modules. This can be a problem because our proposal forces the programmer to split a module in two in this situation, but we do not believe that the need to write more than one instance per type class and data type will be common. The burden of creating a new module is, then, not very severe. Thus, while we lose on expressivity, we gain on simplicity and we think that this is a good trade-off.

Another related work is that on *scoped instances* [2], which suggests a language extension for Haskell that allows instances to be defined inside `let` clauses. An example is given in Figure 19. The proposal suggests choosing the instance that is in the innermost scope, allowing in this scheme also overlapping instances. The proposal does not deal though with the problems of visibility of instances across modules, and thus does not solve the problems of orphan instances nor the problem of pollution of module scopes.

Dreyer, Harper, Chakravarty and Keller have proposed a more radical change to Haskell that allows "viewing type classes as a particular mode of use of modules" [3]. Their work also identifies drawbacks of the current state of the Haskell's type class mechanism — namely, lack of modularity, with consequent inconveniences for the programmer of having always only one instance of a type class for any type, and lack of separation from definition of instances to their availability of use. They also identify a problem of coherence, namely that semantics might differ based on a decision of overloading resolution made by the type inference algorithm. Their solution is to require that the scope of instances be confined

Fig. 17. The function `incoming` as it is on [1, section 5.3], for reference.

```
incoming
  | isHiding = exps
      `minusRel` listed
  | otherwise = listed
```

Fig. 18. The new `incoming` function that also deals with instances.

```
incoming
  | isHiding = exps
      `minusRel` listed
  | otherwise = unionRels
      [listed, instances exps]
```

to the global module level, where required type annotations identify whether overloading has been resolved and, if not, the set of permissible instances. In our proposal, as in Haskell, instances are always at the global module level (our proposal simply allows control of which instances are imported and exported). Overloading resolution is based on the type of the exported instance. If overloading is not resolved, the set of permissible instances is the set of available instances in the importing module.

6 Conclusion

The Haskell language extension proposed in this paper gives more freedom to programmers. On the negative side, this can lead to misuses that may cause programs to become harder to read and to reason about, because assumptions about, for example, the behavior of functions like `sort` may not hold if a non-standard instance of class `Ord` is used. Also, certain operations rely on the presence of some instances, and programmers must be aware of that when redefining instances. Finally, the inclusion of type signatures can change the semantics of a program if such type signatures cause types of exported functions, and instance selection, to be modified. Programmers must then be aware of that and be careful when changing the type of exported entities.

On the positive side, our proposal makes only small changes to the language syntax and semantics. It gives more control to programmers which may construct now programs and libraries that are simpler and more readable. The proposal removes the necessity of the `...By` class of functions and well-known and often discussed problems related with orphan instances. The proposal also makes exportation and importation of instances more homogeneous with other entities, as shown by the fact that the formalization does not need to be changed to deal with instances in our final proposal, but it does need to be changed to handle instances as they are in Haskell nowadays.

6.1 Future work

This paper has presented both syntactic and semantic details of our proposal. An implementation of both syntax alternatives, specifically in the most used Haskell compiler GHC, still needs to be done. The inclusion of a good quality implementation in the main distribution of GHC will allow programmers an

Fig. 19. Example of scoped instance extracted from [2, section 6].

```
e2 = let instance Eq Int where
           x == y = primEqInt (x `mod` 2) (y `mod` 2)
     in 3 == 5
```

opportunity to use the extension on production code, enabling a good evaluation of the utility of the extension in the real world. Rafael Alcântara de Paula is working on implementing this proposal in a Haskell compiler prototype, developed by Rodrigo Ribeiro. The source code of this compiler is available at `https://github.com/rodrigogribeiro/core`.

References

1. Diatchki, I.; Jones, M.; and Hallgren, T.: A formal specification of the Haskell 98 module system. In *Proc. of the 2002 Haskell Workshop*, 2002.
2. Dijkstra, A. et al: Modelling Scoped Instances with Constraint Handling Rules. Universiteit Utrecht. `https://subversion.cs.uu.nl/repos/project.UHC.pub/trunk/pdf/20070406-2213-icfp07-chr-locinst.pdf`.
3. Dreyer, D. et al: Modular Type Classes. In *SIGPLAN Notices*, 42(1):63-70, 2007.
4. Hall, C. V. et al: Type classes in Haskell. In *ACM Transactions on Programming Languages and Systems*. 18(2):109-138, 1996.
5. Kahl, W.; Scheffczyk, J.: Named Instances for Haskell Type Class. In *Preliminary Proc. of the 2001 ACM SIGPLAN Haskell Workshop*. TR UU-CS-2001-62 , Universiteit Utrecht, 2001.
6. Markstrum, S.: Staking Claims: A History of Programming Language Design Claims and Evidence (A Positional Work in Progress). In *Proc. of the Workshop on Evaluation and Usability of Programming Languages and Tools*, 2010.
7. Marlow, S. (editor): Haskell 2010: Language Report. `http://www.haskell.org/onlinereport/haskell2010/`, 2010.
8. Milner, R.; Tofte, M.; Harper, R.: The Definition of Standards ML, version 2. Report ECS-LFCS-88-62, Edinburgh University, Computer Science Dept., 1988.
9. Odersky, M. et al: An overview of the Scala programming language. TR IC/2004/64. École Polytechnique Fédérale de Lausanne, 2004.
10. Peyton-Jones, S.; Jones, M.; Meijer, E.: Type classes: An exploration of the design space. In *Haskell Workshop*, 1997.
11. Pierce, B. C.: Types and Programming Languages. The MIT Press, 2002.
12. Wadler, P; Blott, S.: How to make ad-hoc polymorphism less ad hoc. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages*, pages 60-76, 1989.