

**CONTROLANDO O ESCOPO DE INSTÂNCIAS  
EM HASKELL**



MARCO TÚLIO GONTIJO E SILVA

**CONTROLANDO O ESCOPO DE INSTÂNCIAS  
EM HASKELL**

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: CARLOS CAMARÃO

Belo Horizonte  
Novembro de 2012



MARCO TÚLIO GONTIJO E SILVA

CONTROLLING THE SCOPE OF INSTANCES IN  
HASKELL

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: CARLOS CAMARÃO

Belo Horizonte

November 2012

© 2012, Marco Túlio Gontijo e Silva.  
Todos os direitos reservados.

G641c Gontijo e Silva, Marco Túlio  
Controlling the scope of instances in Haskell / Marco  
Túlio Gontijo e Silva. — Belo Horizonte, 2012  
xxiv, 46 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of  
Minas Gerais

Orientador: Carlos Camarão

1. Type class instances. 2. Modules. 3. Haskell.  
I. Título.

CDU 519.6\*33(043)



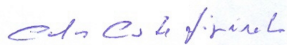
UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO


## FOLHA DE APROVAÇÃO


Controlando o escopo das instâncias em haskell

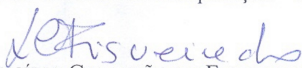
**MARCO TÚLIO GONTIJO E SILVA**

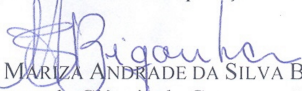
Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

  
PROF. CARLOS CAMARÃO DE FIGUEIREDO - Orientador  
Departamento de Ciência da Computação - UFMG

  
PROF. ANDRÉ LUIS DE MEDEIROS SANTOS  
Departamento de Informática - UFPE

  
PROF. FERNANDO MAGNO QUINTÃO PEREIRA  
Departamento de Ciência da Computação - UFMG

  
PROFA. LUCÍLIA CAMARÃO DE FIGUEIREDO  
Departamento de Computação - UFOP

  
PROFA. MARIZA ANDRADE DA SILVA BIGONHA  
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 20 de dezembro de 2012.





*I dedicate this work to my wife, Ifé.*



# Acknowledgments

I would like to thank everyone that, in some way, contributed to the development of this dissertation. Firstly, I would like to thank the Postgraduate program of the Computer Science Department of UFMG for the opportunity to study and develop my Masters research and this dissertation. I would also like to thank CAPES for the financial support, and Zunnit, UFOP and UNIFOR-MG for letting me work while I was doing my Masters.

I want to thank my parents and my whole family for the support and incentive on my education. I also would like to thank my wife, Priscila, for understanding when I had to study and work on my dissertation and for the incentive and love. I thank also my friend Rafael, who has been a safe haven for me on the whirlwind of life, and someone to discuss from the more practical to the more theoretical and technical aspects of this work.

In a more direct manner, I would like to thank all the teachers from the Computer Science Department that led me here, through the undergraduate course and now in the Masters. In special, my advisor Carlos Camarão, who was always receptive, patient and understanding. I thank him for the incentive and for the suggestions and ideas that were always very interesting and helpful. Finally, I would like to thank some persons that contributed to the dissertation with suggestions of improvements. They are Lucília Camarão, Gláuber Cabral, Atze Dijkstra and Fernando Pereira.



*“Eu quase que nada não sei. Mas desconfio de muita coisa.”*  
(Riobaldo)



# Resumo

O sistema de módulos de Haskell objetiva a simplicidade e possui a notável vantagem de ser fácil de aprender e usar. Entretanto, instâncias de classes de tipo em Haskell são sempre exportadas e importadas entre módulos. Isso quebra a uniformidade e simplicidade do sistema de módulos e introduz problemas práticos. Instâncias criadas em módulos diferentes podem conflitar umas com as outras e podem fazer com que seja impossível importar dois módulos que contenham definições de uma mesma instância se essa instância for utilizada. Isso faz com que seja muito inconveniente a definição de duas instâncias diferentes da mesma classe de tipos para o mesmo tipo em diferentes módulos de um mesmo programa. A definição de instâncias em módulos onde nem o tipo nem a classe de tipos são definidos se tornou uma má prática, e essas instâncias foram chamadas de instâncias órfãs. Somente esse tipo de instância pode causar conflitos já que, se instâncias forem definidas apenas no mesmo módulo do tipo ou da classe de tipos, só poderá existir uma instância para cada par de classe e tipo.

Nessa dissertação nós apresentamos e discutimos uma solução para esses problemas que simplesmente permite que haja controle sobre a importação e exportação de instâncias entre módulos, por meio de uma pequena alteração na linguagem. A solução é apresentada em duas versões. A versão final, mais consistente, não é compatível com Haskell, isto é, programas que funcionam em Haskell podem deixar de funcionar com essa alteração. A versão intermediária traz os benefícios da proposta, é compatível com Haskell, mas é um pouco menos consistente. Para evitar que o programador precise escrever nomes de instâncias muito longos nas listas de controle de importação e exportação de módulos, propomos outra pequena alteração na linguagem, que torna possível dar nomes mais curtos a instâncias.

Também mostramos como a especificação formal do sistema de módulos precisa ser adaptada para lidar com nossa proposta. Como a especificação formal não tratava instâncias, primeiro adaptamos essa especificação para tratar instâncias e, em seguida, mostramos como nossa proposta é especificada formalmente.

**Palavras-chave:** Instâncias de classes de tipo, Módulos, Haskell.



# 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. Instances created in different modules can conflict with each other, and can make it impossible to import two modules that contain the same instance definitions if this instance is used. Because of this, it is very inconvenient to define two distinct instances of the same type class for the same type in a program. The definition of instances in modules where neither the data type nor the type class are defined, called orphan instances, became a bad practice. Only these instances can cause conflicts since, if instances are defined in the same module of the type or of the type class, only one instance can possibly exist for each pair of class and type.

In this dissertation we present and discuss a solution to these problems that simply allows control over importation and exportation of instances between modules, through a small change in the language. The solution is presented in two versions. The final version, more consistent, is not compatible with Haskell, that is, Haskell programs may not work with this change. The intermediate version, on the other hand, brings the benefits of the proposal while being compatible with Haskell, but it is less consistent. In order to avoid very long names for instances in module importation and exportation control lists, we propose another small change in the language to make it possible to give shorter names to instances.

We also show how a formal specification of the module system must be adapted to include our proposal. As the formal specification didn't handle instances in general, we first adapt this specification to handle instances, and then show how our proposal can be formally specified.

**Palavras-chave:** Type class instances, Modules, Haskell.



# List of Figures

1.1	Definition of class <code>Eq</code> from the Haskell Prelude. . . . .	2
1.2	<code>Bool</code> data type as defined in the Haskell Prelude. . . . .	2
1.3	Instance of type class <code>Eq</code> for type <code>Bool</code> . . . . .	2
1.4	A simple module declaration. . . . .	4
1.5	Abstract data types implemented through the module system. . . . .	4
1.6	A module with an empty export list. . . . .	4
1.7	Basic import clause. . . . .	5
1.8	Import clause with import list. . . . .	5
1.9	Disambiguation using import lists. . . . .	5
1.10	Disambiguation using the keyword <code>hiding</code> . . . . .	5
1.11	Disambiguation using the module name. . . . .	6
1.12	Renaming of modules using the keyword <code>as</code> . . . . .	6
1.13	Qualified import to avoid conflicting names. . . . .	7
2.1	Module <code>T</code> . . . . .	10
2.2	Module <code>D</code> . . . . .	10
2.3	Module <code>I1</code> . . . . .	10
2.4	Module <code>I2</code> . . . . .	11
2.5	<code>Main</code> module of the example of orphan instances. . . . .	11
2.6	Import chain for the example of orphan instances. . . . .	11
2.7	Example of the usage of <code>newtype</code> to create a new instance. . . . .	12
2.8	Function <code>addNumber</code> using type classes. . . . .	13
2.9	Function <code>addNumber</code> using functions as parameters. . . . .	13
2.10	Generic map using a type class and an associated data type. . . . .	16
3.1	New syntax for the export clause. . . . .	18
3.2	New syntax for the import clause. . . . .	18
3.3	Second version of Module <code>I1</code> , using the proposed extension. . . . .	22

3.4	Second version of the <code>Main</code> module, using the proposed extension. . . . .	22
3.5	Module <code>Definition</code> , used in the example of unexpected behavior that arises from misuse of local instances. . . . .	24
3.6	<code>Main</code> module of the example of unexpected behavior that arises from misuse of local instances. . . . .	24
4.1	Original code for the call of <code>writeIface</code> . . . . .	28
4.2	New code for the call of <code>writeIface</code> . . . . .	28
4.3	The code of the function <code>filterEl</code> . . . . .	28
4.4	The original code of the function <code>readIface</code> . . . . .	29
4.5	The new code of the function <code>readIface</code> . . . . .	29
4.6	The new code of the function <code>readIface</code> . . . . .	30
4.7	The original code of the instance check. . . . .	30
4.8	The new code of the instance check which does not allow more than one instance of the same class for the same type. . . . .	30
4.9	Function <code>filterI1</code> on the intermediate version. . . . .	31
4.10	Function <code>filterInstancesI1</code> on the intermediate version. . . . .	32
4.11	Function <code>filterI1</code> on the final version. . . . .	32
4.12	Function <code>doFilterI1</code> on the final version. . . . .	33
4.13	Function <code>filterInstancesI1</code> on the final version. . . . .	33
4.14	Function <code>filterEl</code> filtering instances. . . . .	33
4.15	Function <code>filterInstancesEl</code> . . . . .	34
5.1	Auxiliary functions for filtering instances in the module system. . . . .	37
5.2	Function <code>exports</code> as in [Diatchki et al., 2002, Section 5.2]. . . . .	39
5.3	New function <code>exports</code> . . . . .	39
5.4	The function <code>incoming</code> as it is on [Diatchki et al., 2002, Section 5.3], for reference. . . . .	39
5.5	The new <code>incoming</code> function that also deals with instances. . . . .	39
6.1	Example of scoped instance extracted from [Dijkstra et al., 2007, Section 6].	42

# List of Tables

3.1	The semantics translation from the intermediate syntax to the final. . . . .	20
-----	--	----



# Contents

<b>Acknowledgments</b>	<b>xi</b>
<b>Resumo</b>	<b>xv</b>
<b>Abstract</b>	<b>xvii</b>
<b>List of Figures</b>	<b>xix</b>
<b>List of Tables</b>	<b>xxi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Type classes . . . . .	1
1.2 Modules . . . . .	3
1.3 Objectives of this dissertation . . . . .	6
1.4 Contributions . . . . .	7
1.5 Outline of this work . . . . .	8
<b>2 Background</b>	<b>9</b>
2.1 Defining special purpose instances . . . . .	9
2.2 Orphan instances . . . . .	14
2.2.1 Real world cases . . . . .	15
2.3 Conclusion . . . . .	16
<b>3 Solution</b>	<b>17</b>
3.1 Final alternative . . . . .	18
3.2 Intermediate alternative . . . . .	20
3.3 Instance names . . . . .	21
3.4 Instance scope . . . . .	21
3.5 Problems and Solutions . . . . .	23
3.6 Conclusion . . . . .	26

<b>4</b>	<b>Implementation</b>	<b>27</b>
4.1	Export and import lists . . . . .	28
4.2	Conflicting instances . . . . .	29
4.3	Intermediate version . . . . .	31
4.4	Final version . . . . .	31
4.5	Testing . . . . .	34
4.6	Conclusion . . . . .	35
<b>5</b>	<b>Extending the Module System specification</b>	<b>37</b>
5.1	Haskell and the intermediate alternative . . . . .	38
5.2	The final alternative . . . . .	40
<b>6</b>	<b>Related work</b>	<b>41</b>
<b>7</b>	<b>Conclusion</b>	<b>43</b>
7.1	Contributions . . . . .	43
7.2	Future work . . . . .	44
	<b>Bibliography</b>	<b>45</b>



# Chapter 1

## Introduction

Modern programming languages are using more flexible type systems in order to accept a larger set of programs. The goal of these type systems is to reject less programs that would work correctly but that would not be accepted by more restrictive type systems. One of the techniques to achieve this flexibility is to 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 [Wadler and Blott, 1989], which supports 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.

In C++ this kind of polymorphism is achieved by means of overloading function names [Stroustrup, 1997, Section 7.4]. From the compiler perspective, two functions with the same name but with different types as the parameter are considered two different functions. When there are more than one option of function to call for a specific type, or for a polymorphic symbol, like numerals, the compiler applies a set of rules to decide between them. In order to avoid the need to understand the context to define which of the functions with the same name were called, “return types are not considered in overloading resolution” [Stroustrup, 1997, Section 7.4.1].

### 1.1 Type classes

Haskell is a programming 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<sup>1</sup>. Type classes are a language mechanism

---

<sup>1</sup><http://industry.haskell.org/>

**Figure 1.1.** Definition of class `Eq` from the Haskell Prelude.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  x == y = not (x /= y)
  x /= y = not (x == y)
```

**Figure 1.2.** `Bool` data type as defined in the Haskell Prelude.

```
data Bool = False | True
```

**Figure 1.3.** Instance of type class `Eq` for type `Bool`.

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
```

that was introduced in Haskell for supporting ad-hoc polymorphism [Hall et al., 1996]. A type class specifies a set of overloaded names together with type annotations for them. For instance, the names defined for class `Eq`, from the Haskell Prelude are given in Figure 1.1. All mentions to the Haskell Prelude in this dissertation are a reference to the definition of the Prelude given on the Haskell Report [Marlow, 2010]. As can be seen in this figure, the definition of the class can contain a standard implementation of some methods, and this implementation can be based on other methods of the same class.

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. As an example, a possible definition of an instance of `Eq` for `Bool` is shown in Figure 1.3. The definition of the data type `Bool`, from the Haskell Prelude [Marlow, 2010] is shown in Figure 1.2.

A type class declaration defines overloaded names, also called class *members*, with corresponding types, and an instance declaration gives a value for each class member, referred to as a *member value*, sometimes also referred to in the literature as a “member function”.

The standard semantics of Haskell is based on the application of so-called *dictionaries* to overloaded names [Wadler and Blott, 1989; M. Jones, 1994; Hall et al., 1996]. A *dictionary* is a tuple that corresponds to an instance declaration, and contains values that correspond to the definitions given in the instance declaration for each class member. A dictionary of a superclass contains also a pointer to a dictionary of each of its subclasses [M. Jones, 1994; Hall et al., 1996; Faxén, 2002]. The dictionary-passing semantics is fully based on the syntax of constrained types, which is not changed in any way by our proposal.

## 1.2 Modules

A module system of a programming language is intended to provide support for a modular construction of software systems. In some languages the module system provides a type-safe abstraction mechanism, where definitions can be parameterized so that modules can be instantiated for different kinds of entities. This is the case for example of Standard ML [Milner et al., 1988].

A module system can also merely allow a program to be divided into parts that can be compiled separately, such as in Java [Arnold et al., 2006] or Scala [Odersky et al., 2012]. 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 [Marlow, 2010, Chapter 5].

The Haskell module system aims for simplicity [Hudak et al., 2007, Section 8.2] and has the notable advantage of being easy to learn and use. Declaring a module is as simple as shown in Figure 1.4. The system provides control over exported entities through export lists, which is used to construct abstract data types. Only the entities named on the list, and all type class instances, are exported from the module. In the module of Figure 1.5, the constructor of the data type is not exported, and the type content can only be accessed through the exported functions. This provides a way of supporting abstract types in the language. If the export list is absent, all entities defined in the module will be exported, as in Figure 1.4. The export list can be empty. In this case, only type class instances defined in the module are exported. For instance, in Figure 1.6, only `instance Num Char` is exported. `foo` can only be used inside Module M3.

Importing a module is as simple as defining a module. The basic import clause is shown in Figure 1.7. The names of the modules follow a hierarchy to provide a

**Figure 1.4.** A simple module declaration.

```
module M1 where
```

**Figure 1.5.** Abstract data types implemented through the module system.

```
module M2 (IncInt, zero, inc) where

newtype IncInt = CIncInt Integer

zero :: IncInt
zero = IncInt 0

inc :: IncInt -> IncInt
inc (IncInt i) = IncInt $ succ i
```

**Figure 1.6.** A module with an empty export list.

```
module M3 () where

foo :: Int
foo = 3

instance Num Char where
  // ...
```

better organization of modules. This hierarchy is not strictly defined in a global sense. Imports can also have an import list, which controls which entities exported by the module are being imported. Import lists are very important for avoiding name conflicts in a module. Only entities mentioned in the import list, and all type class instances, are imported. In the example of Figure 1.8, `IncInt` and `zero` are imported, but not `inc`, defined in Module M2, Figure 1.5.

To avoid name conflicts, it is possible to create an import list. For instance, name `empty` is defined in both modules `Data.Set` and `Data.Map`. So, if only `empty` from `Set` is used in a module, and other functions from Module `Data.Map` are used, but not `empty`, it is possible, using import lists, not to import `empty` from `Data.Map`. This is shown in Figure 1.9. If the number of names not to be imported is bigger

**Figure 1.7.** Basic import clause.

```
import Data.Char
```

**Figure 1.8.** Import clause with import list.

```
import M2 (IncInt, zero)
```

**Figure 1.9.** Disambiguation using import lists.

```
import Data.Set
import Data.Map (singleton, insert)

emptySet :: Set a
emptySet = empty
```

than the number of names to be imported from one module, the keyword `hiding` can be used to specify which names not to import. An example is shown in Figure 1.10. However, it may be the case that both `empty`'s are used in a module. A way to solve this kind of conflict is to prefix the name of an imported entity with the module name. If both modules are imported, `empty` from one module can be distinguished from that of another module by prefixing the name with the module name, as shown in Figure 1.11.

Module names can get quite big. For instance, package `vector`, which is part of the default distribution of the most used Haskell compiler, the Glasgow Haskell Compiler, has a module called `Data.Vector.Fusion.Stream.Monad.Safe`. To avoid

**Figure 1.10.** Disambiguation using the keyword `hiding`.

```
import Data.Set
import Data.Map hiding (empty)

emptySet :: Set a
emptySet = empty
```

**Figure 1.11.** Disambiguation using the module name.

```
import Data.Set
import Data.Map

emptySet :: Set a
emptySet = Data.Set.empty

emptyMap :: Map a b
emptyMap = Data.Map.empty
```

**Figure 1.12.** Renaming of modules using the keyword `as`.

```
import Data.Set as S
import Data.Map as M

emptySet :: Set a
emptySet = S.empty

emptyMap :: Map a b
emptyMap = M.empty
```

such big names on each usage of a symbol, modules can be renamed using the keyword `as`. Figure 1.12 shows an example of such renaming.

In Figures 1.11 and 1.12, all occurrences of `empty` must be prefixed with the module name. Sometimes, a module has a lot of uses of one of the conflicting names, and very few uses of the others. In addition, it may happen that this is the case for a lot of names in the module. In this case, a qualified import is a good option. An example is shown in Figure 1.13. Notice that module name of `Data.Set` is not needed in the use of `empty`. The combination of qualified imports with renaming with keyword `as` is a very common pattern.

### 1.3 Objectives of this dissertation

The simplicity of the module system is partly hindered by the special treatment given to the scope of instances, for which there is no control on exportation and importation. As defined in the Modules chapter of the Haskell 2010 Report [Marlow, 2010, Section 5.4],

**Figure 1.13.** Qualified import to avoid conflicting names.

```
import Data.Set
import qualified Data.Map

emptySet :: Set a
emptySet = empty

emptyMap :: Map a b
emptyMap = Data.Map.empty
```

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 for a module to import two modules such that each one defines an instance of the same type class to the same data type, if the importing module, or any module that imports it, uses the instance. This happens both if the definitions are different or the same in the different modules. 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 disallow reasonable and correct code. Furthermore, many 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 dissertation we propose an extension to Haskell that 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.

## 1.4 Contributions

This dissertation proposes an extension for the Haskell programming language that solves the problem of orphan instances and enables an easy way to create special purpose instances. This extension is proposed in two different syntax alternatives. The first, called *intermediate*, is backwards compatible but is not very intuitive. The second, called *final*, is very simple and intuitive, but is not backwards compatible.

## 1.5 Outline of this work

Chapter 2 illustrates how the absence of control of the visibility of instances makes it hard or impossible to use instances for a certain type with a special purpose, in Section 2.1, and describes orphan instances, an instance that is defined neither in the module that defines the type class nor in the module that defines the data type, in Section 2.2.

In Chapter 3 we present our proposal, with two possible alternatives: the final one, in Section 3.1, and the intermediate one, in Section 3.2. This chapter also presents a complementary proposal that gives names to instances, in Section 3.3. Some consequences of the control over the scope of instances are discussed in Section 3.4. Problems that can occur by the adoption of our proposal and possible solutions to them are presented and discussed in Section 3.5.

The implementation of the proposal is detailed in Section 4.

Chapter 5 describes one way of extending a published formalization of Haskell's module system [Diatchki et al., 2002] to handle our proposal. As instances were not modeled on the original formalization, they are handled at first, and then our proposal is included.

Chapter 6 describes related work and Chapter 7 concludes the dissertation.



# Chapter 2

## Background

This chapter illustrates how the absence of control of the visibility of instances makes it hard or impossible to use instances for a certain type with a special purpose, in Section 2.1, and describes orphan instances, an instance that is defined neither in the module that defines the type class nor in the module that defines the data type, in Section 2.2.

### 2.1 Defining special purpose instances

As instances are always exported and imported, all instances defined in a program are visible at the program's topmost module, that is, the `Main` module. If two instances of the same type class for the same data type are defined in different parts of the program, they will be visible in at least this `Main` module. The number of modules in which the two instances are visible can be though much bigger. In these modules any use of one of these instances will result in a compilation error. In these modules it is impossible to use an overloaded function of one of these instances, even if the program explicitly determines which instance is used. Because of this restriction, although it is possible to use more than one instance of a type class for a given type in a program, in some cases this can be rather inconvenient, since the use of an overloaded function for a given type would not be possible in some parts of the program. In addition, this is not useful, since each polymorphic function that uses such an instance has to be instantiated and thus cannot be used as a polymorphic function.

For example, overloaded function `g`, defined in Figure 2.5 cannot be used in the `Main` module. Either `g1`, defined in Module `I1`, Figure 2.3, or `g2`, defined in Module `I2`, Figure 2.4, would have to be used. The instantiated version of each polymorphic function that uses one of the overloaded definitions from the type class has to be instantiated in the same module that defines the instance. Also, it is not possible to

**Figure 2.1.** Module T.

```

module T where

class T a where
  t :: a

g :: T a => (a, a)
g = (t, t)

```

**Figure 2.2.** Module D.

```

module D where

data D = D

```

**Figure 2.3.** Module I1.

```

module I1 where

import T
import D

instance T D where
  t = undefined

g1 :: (D, D)
g1 = g

i1 :: a
i1 = undefined

```

use any overloaded function defined in Modules I1 or I2. Figure 2.6 gives an overview of the import chain of the modules involved in this example. These are significant disadvantages for the use of type classes.

Due to the inconvenience of defining and using more than one instance for a given type, the programmer will not be able, for example, to sort values of a given type by using two different techniques, applying an overloaded function `sort`. More specifically,

**Figure 2.4.** Module I2.

```

module I2 where

import T
import D

instance T D where
  t = undefined

g2 :: (D, D)
g2 = g

i2 :: a
i2 = undefined

```

**Figure 2.5.** Main module of the example of orphan instances.

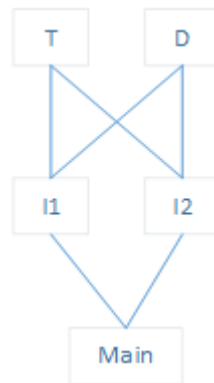
```

import I1
import I2

f :: a -> a -> a
f = undefined

h :: a
h = f i1 i2

```

**Figure 2.6.** Import chain for the example of orphan instances.

**Figure 2.7.** 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)

```

a programmer cannot use case-sensitive ordering to sort a list of strings in a part of a program and case-insensitive ordering in another.

A general way to work with these problems is to create a new encapsulated data type, using `newtype`, and define a different instance for it. The example in Figure 2.7 illustrates this solution. This works, but it is verbose and not efficient. In other words, it is “too clunky”<sup>1</sup>. 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 each specific problem such as this one, 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 reasonable since it would add parameters in lots of cases, making the code more complicated. In addition, it goes against the idea of making code simpler and more reusable by means of overloading.

<sup>1</sup>In Lennart Augustsson’s words.  
a-world-without-orphans/#comment-609

<http://lukepalmer.wordpress.com/2009/01/25/>

**Figure 2.8.** Function `addNumber` using type classes.

```

addNumber :: (Ord a, Num a, Show a) => a -> String -> String
addNumber n s
  | n < 1000 = s ++ show n
  | otherwise = s

```

**Figure 2.9.** Function `addNumber` using functions as parameters.

```

addNumber
  :: (a -> a -> Bool)
    -> (Integer -> a)
    -> (a -> String)
    -> a
    -> String
    -> String
addNumber lt fi sh n s
  | n 'lt' fi 1000 = s ++ sh n
  | otherwise = s

```

For instance, suppose you have a function called `addNumber` that receives a numeric value, which is represented by type class `Num` in Haskell. This type class has a method called `fromInteger` that is used to convert from a numeric literal to any of the types that instantiate this class. The function `addNumber` will receive the numeric value and a string. If the number is smaller than 1.000, it will concatenate the number as a string to the given string. Otherwise, it will return the string as it is. The code for this function is on Figure 2.8.

To generalize this function so that it could be used with another type of ordering on an specific data type, at least three different functions would have to be passed to it: one to tell how to compare a value with another, one about how to convert from `Integer` to the given type and one to convert the given type to a string. As it can be seen in Figure 2.9, the code is much harder to read. This is a simple example that will get much worse with classes that are more complicated. The more generic the code is, the more functions would have to be passed as a parameter.

## 2.2 Orphan instances

To define an instance, both the type class and the data type must be in scope. There are four possibilities for that to happen:

1. the data type and the type class are defined in the same module of the instance;
2. the type class is defined in the same module of the instance, but the data type is imported;
3. the data type is defined in the same module of the instance, but the type class is imported;
4. both the data type and the type class are imported by the module that defines the instance.

Defining two instances of the same type class for the same data type in a module will create a compilation error. In addition, if an instance is imported and another one is defined for the same pair of class and type, there will be a compilation error. As the instances are always exported and imported, so all modules that import the type class definition and the data type definition, what is necessary to define a new instance, will also import the already defined instance. Therefore, in Case (1) it is impossible to have more than one instance without compilation errors.

In Case (2), it is not possible that the imported module containing the data type will already have an instance for that pair of class and type, because the class is not yet in scope in that module, since it is defined in the importing module. Similarly to Case (1), if another module imports the type class, it will necessarily import the instance, what would make it impossible for it to define a new one. The same happens with Case (3), where the data type is defined in the same module. In both cases, it is not possible to have more than one instance for the same pair of class and type.

In Case (4), both the type class and the data type are imported by the module that defines the instance, so that it is possible that another module will also import these definitions and create another instance. This is the only case where it is possible to create two instances of the same pair of class and type. Therefore, 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.

As there is no control over the importation and exportation of instances, these instances are problematic and are called *orphan instances*. Orphan instances are instances defined in a module that contains neither the definition of the data type nor

the definition of the type class. Orphan instances thus enable the creation of distinct instances of a type class for the same data type.

They are especially troublesome when a module defines other functions that are not related with the instance. For example, if we have a Module `T`, shown in Figure 2.1, that defines a type class `T`, a Module `D`, shown in Figure 2.2, that defines a data type `D`, and two Modules `I1`, shown in Figure 2.3, and `I2`, shown in Figure 2.4, that define instances of `T` for `D`, we would not be able to import both `I1` and `I2` in the same module, if this module uses `f`, or a function overloaded on class `T`.

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` in 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. If the `Main` module, or some other module where both instances are available, uses `f`, or a function overloaded on `T`, it will not be possible to import `I1` and `I2` in the same program. Modifying `I1` or `I2` is not always possible in practice because they may be part of a third-party library.

### 2.2.1 Real world cases

It is worth noticing that the problems enumerated in the last sections are not only potential problems. They happen in real world uses of the language. For example, the instance of type class `Monad` for data type `Either` is defined neither in the module where `Monad` is defined, `Control.Monad`, nor in the module where `Either` is defined, `Data.Either`. It is defined instead in Module `Control.Monad.Error`, from package `mtl`, and also in Module `Control.Monad.Trans.Error`, from package `transformers`<sup>2</sup>. If these two modules are directly or indirectly imported by a module, it would not be possible to use this instance on this module.

Lennart Augustsson presented “a concrete example” of a case where orphan instances would be desirable<sup>3</sup>. There are libraries from pretty printing data types, with

---

<sup>2</sup>This example is on the wiki page at [http://www.haskell.org/haskellwiki/Orphan\\_instance](http://www.haskell.org/haskellwiki/Orphan_instance)

<sup>3</sup><http://lukepalmer.wordpress.com/2009/01/25/a-world-without-orphans/#comment-601>

**Figure 2.10.** Generic map using a type class and an associated data type.

```
class Unbox k v where
  data Map k v :: *
  empty :: Map k v
  lookup :: k -> Map k v -> Maybe v
  insert :: k -> v -> Map k v -> Map k v
```

type classes that include “instances for all relevant prelude types”. There are also packages with a data type for dealing with JSON data. It would be good to write an instance for pretty printing JSON data in yet another package, since those packages are unrelated and should not necessarily know about each other and implement instances about the other. But, if this instance is written in another package, it will be an orphan instance, which would cause the problems described in Section 2.2.

Johan Tibell proposes a generic map implementation using a type class for the common map functions and an associated data type for that type class<sup>4</sup>. The code proposed is shown in Figure 2.10. This would require an instance for each pair of key and value of the map. Many instances would have to be generated for the prelude types, probably using Template Haskell, a mechanism for automatic code generation. Even so, for library types the user would have to generate the instance. This way, it would not be possible for the user to generate the instance of the data type defined in another package without creating an orphan instance.

## 2.3 Conclusion

Orphan instances are an important problem for which many special treatments are required by compilers. In addition, creating a special purpose instance is not a very uncommon situation and it would be good to have a simple, practical and intuitive way to do it. The next chapter will present a solution to both of these problems.

---

<sup>4</sup><http://www.haskell.org/pipermail/glasgow-haskell-users/2010-August/019052.html> .



# Chapter 3

## Solution

We propose that instances should be exportable and importable. It is a natural, simple proposal that has already been mentioned<sup>1</sup>, but this work provides a detailed description and discussion, including required changes in the language definition.

The proposal eliminates 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, since 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 desired 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.

If adopted, these alternative proposals should preferably be enabled by compilers by the use of a compilation flag. There should exist then a different flag for each proposal.

In both cases, `export` and `import` clauses used in The Haskell 2010 Report [Marlow, 2010, Sections 5.2 and 5.3] are changed to have a new option, with the header of an instance declaration [Marlow, 2010, Section 4.3.2]: `instance [scontext =>] qtycls`. The option identifies whether an instance should be exported, imported or hidden. `import` and `export` clauses with the new option are defined as in Figures 3.1 and 3.2.

---

<sup>1</sup>By Yitzchak Gale on Stack Overflow <http://stackoverflow.com/questions/3079537/orphaned-instances-in-haskell/3079748#3079748> .

**Figure 3.1.** New syntax for the export clause.

<code>export</code>	$\rightarrow$	<code>qvar</code>	
		<code>tycons [(..) (cname<sub>1</sub>, ..., cname<sub>n</sub>)]</code>	$(n \geq 0)$
		<code>tycls [(..) (var<sub>1</sub>, ..., var<sub>n</sub>)]</code>	$(n \geq 0)$
		<code>module modid</code>	
		<code>instance [scontext =&gt;] qtycls</code>	

**Figure 3.2.** New syntax for the import clause.

<code>import</code>	$\rightarrow$	<code>var</code>	
		<code>tycon [(..) (cname<sub>1</sub>, ..., cname<sub>n</sub>)]</code>	$(n \geq 0)$
		<code>tycls [(..) (var<sub>1</sub>, ..., var<sub>n</sub>)]</code>	$(n \geq 0)$
		<code>instance [scontext =&gt;] qtycls</code>	

### 3.1 Final alternative

In the final alternative instances are imported and exported just as other entities in Haskell. There are ten distinct cases where import clauses are affected by the proposal, presented below by considering the example shown previously in Figure 2.3, similarly to [Marlow, 2010, Section 5.3.4]:

1. `import I1` imports everything from Module `I1`, including instances, as occurs currently in Haskell;
2. `import I1 ()` imports nothing, as occurs if this line is commented or absent;
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.
6. `import qualified I1` makes all definitions accessible in a qualified manner, but not the instances;
7. `import qualified I1 ()` imports nothing, as occurs if this line is commented or absent, as in Item (2);
8. `import qualified I1 (instance T D)` imports only the instance, as in Item (3);

9. `import qualified I1 hiding (instance T D)` makes all definitions accessible in a qualified manner, but not any of the instances, including the one mentioned, as in Item (6);
10. `import qualified I1 (i1)` makes only `i1` accessible in a qualified manner.

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.

Usually, qualified imports are done as in Item (6), because there is not much advantage of restricting the imported entities as in Item (10), if you must specify the module on each usage. Items (7), (8) and (9) are not very used because they have the same semantics of other items, which are simpler. They are (2), (3) and (6) respectively. In Haskell 2010 instances are imported even if the import is qualified. Qualified imports were described with detail in Section 1.2. This has the undesirable property that the inclusion of a qualified import in a module can create a compilation error of conflicting instances. In the final alternative of our proposal, the inclusion of a qualified import does not import instances. If they should be imported, they must be mentioned in an import list. This provides the property that the inclusion of a qualified import in a module will not bring any compilation errors. In addition, it is more consistent with the way qualified imports work with other entities: they do not affect the general scope of the importing module, they only create a way to access the entities from the imported module. If instances were to be imported in qualified imports, as they are in Haskell 2010, the general scope of the module would be affected.

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

11. `module I1 where` exports everything in `I1`, including the instance, as occurs currently in Haskell;
12. `module I1 () where` exports nothing, not even the instance;
13. `module I1 (instance T D) where` exports only the instance, such as `module I1 () where` in Haskell 2010;
14. `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 clause given in (2), (5), (7), (10), (12) or (14) is correct in Haskell 2010, but has a different meaning than the one we are proposing. In Haskell 2010, the instance is imported or exported but in our proposal, it is not. In our view this language extension should be incorporated in the language in a second step, after the adoption of the intermediate alternative, described next.

**Table 3.1.** The semantics translation from the intermediate syntax to the final.

	<b>Intermediate (or Haskell 2010)</b>	<b>Final</b>
2	<code>import I1 ()</code>	<code>import I1 (instance T D)</code>
5	<code>import I1 (i1)</code>	<code>import I1 (i1, instance T D)</code>
7	<code>import qualified I1 ()</code>	<code>import qualified I1 (instance T D)</code>
10	<code>import qualified I1 (i1)</code>	<code>import qualified I1 (i1, instance T D)</code>
12	<code>module I1 () where</code>	<code>module I1 (instance T D) where</code>
14	<code>module I1 (i1) where</code>	<code>module I1 (i1, instance T D) where</code>

## 3.2 Intermediate alternative

The intermediate alternative differs from the final alternative, just to be backwards compatible. In Items (2), (5), (7), (10), (12) and (14) instances are imported or exported. The only way to avoid an instance from being imported is by using keyword `hiding` in an import list. There is no way to avoid an instance from being exported. In the intermediate alternative, mentions of instances in the import and export lists are not considered if they are not on the hiding list. Therefore, (13) is valid and has the same effect as (12), and the same goes for Items (3) and (8).

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 in Table 3.1. As the intermediate alternative has a syntax that is backwards compatible with Haskell 2010, Table 3.1 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), (10), (12) and (14) should be considered as bad programming practice. These should be gradually replaced by their final version, as shown in Table 3.1. 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), (10), (12) and (14) — should be acceptable, but they will have the semantics defined here, and not the old semantics.

New languages, in order to justify their existence, make claims that fall under three categories [Markstrum, 2010, p. 1]: “novel features, incremental improvement on existing features, and desirable language properties”. This work 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)`.

Although it has a similar name, the Named Instances proposal [Kahl and Schefczyk, 2001] 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 in Chapter 6.

### 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 3.3, 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`.

**Figure 3.3.** 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

```

**Figure 3.4.** 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

```

If the `Main` module, shown in Figure 3.4, 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 the `Main` module. As already stated, if the line is commented, the instance defined in `I2` will be used.

Some instances may have a context, like `instance Eq a => Eq [a]`. So, to be able to use the functions from class `Eq` on, for instance, `[Int]`, the module will have to

import both `instance Eq [a]` and `instance Eq Int`. The importation of `instance Eq [a]` does not bring into scope the instances required by the context of this instance.

## 3.5 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” [Peyton Jones et al., 1997, 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 [Pierce, 2002, p. 3], and this is our motivation.

On the other hand, there are reasons why the proposed mechanism was not included in the language in the first place. It may be argued that changing the definition of an instance in a program makes it harder to understand what the code means. This is only a problem if the changes made 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 create unintuitive expressions like `let 1 + 1 = 3 in 1 + 1`, which redefines a function denoted by `(+)` in local scope, without properly changing the related type class or its instances. Therefore, this is not going to be the only case in the language where basic constructs 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`, show in Figure 3.5, `'a'` is inserted after `'B'`, since in case-sensitive order it comes later. Suppose `iCmp` is a comparison function for case-insensitive values of type `Char`. The call of `member` in the `Main` module shown in Figure 3.6 searches for `'a'` before `'B'`, because it uses the case-insensitive ordering, 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.

Also, it is possible to do the same thing in Haskell, if the conflicting instance

**Figure 3.5.** 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

```

**Figure 3.6.** 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

```

is not used in the modules that imports the definitions. To avoid using the instance in this module, each module that defines a different instance can provide a function that has the same behavior of the overloaded function, but already instantiated for a given type. For instance, both modules could provide a function `insert'`, with type `Char -> Set Char -> Set Char`, instead of the generic type from `insert`, which is a `-> Set a -> Set a`. The same can be done for `member`. In this case, the modules that import the definitions would not have to use the overloaded functions; instead, it would use only the already instantiated functions. Therefore, no type errors will exist, and the behavior of the program will be equal to the one presented in the last example, without using our proposed extension<sup>2</sup>.

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.<sup>3</sup> Although this is in general un-

<sup>2</sup>There is an example showing how to do so, with source code, in <https://gist.github.com/3854294>.

<sup>3</sup>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>.



desirable, 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<sup>4</sup> and an “a *really* amazing example”<sup>5</sup> using polymorphic recursion<sup>6</sup>. We believe that the advantages of our proposal outweigh disadvantages related to these issues.

---

<sup>4</sup>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> .

<sup>5</sup>As mentioned by Simon Peyton-Jones in <http://www.haskell.org/pipermail/haskell/2001-May/007133.html> .

<sup>6</sup>Described by Lennart Augustsson in <http://www.haskell.org/pipermail/haskell/2001-May/007122.html> .

## 3.6 Conclusion

The proposal solves in a simple way important problems that are currently present in Haskell. The simplicity of the implementation of this proposal, shown in the next chapter, provides yet another reason for integrating this proposal to Haskell.

# Chapter 4

## 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 cannot 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 implementation of our proposal will require removing elements from this list while importing and exporting definitions from a module.

The creation of dictionaries used to implement type classes is not affected. They are created in the same place and at the same time as without our proposal. The number of available dictionaries, on the other hand, can be reduced, since not all dictionaries defined on imported modules are necessarily available for use on the `Main` module. This may lead to an improvement in performance of the generated binary.

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.

The implementation of the proposal was done in a prototype developed by Rodrigo Ribeiro<sup>1</sup>. This prototype contains the front-end of a Haskell compiler. It does not generate executable binaries, it only performs the type inference and creates an interface file for each module, with all defined symbols together with their types. At the beginning of the work, the prototype did not take import and export lists into consideration. Therefore, the first step of the work was to make the prototype handle

---

<sup>1</sup>The prototype is available at <https://github.com/rodrigogribeiro/mptc> and our implementation is available at <https://github.com/marcotmarcot/mptc> . Our implementation uses an adapted version of the library `haskell-src-exts`. The original library can be found at <http://hackage.haskell.org/package/haskell-src-exts> and our adapted version can be found at <https://github.com/marcotmarcot/haskell-src-exts-scoped-instances> .

**Figure 4.1.** Original code for the call of `writeIface`.

```
writeIface dir' m' ifaces miface
```

**Figure 4.2.** New code for the call of `writeIface`.

```
writeIface dir' m' ifaces $ filterEl m miface
```

**Figure 4.3.** The code of the function `filterEl`.

```
filterEl :: Module -> Iface -> Iface
filterEl (Module _ _ _ _ Nothing _ _) i = i
filterEl
  (Module _ _ _ _ (Just exs) _ _)
  i@(Iface {synonyms = s, classes = c, assumps = a})
  = i
  {synonyms = doFilterEl exs s,
   classes = doFilterEl exs c,
   assumps = doFilterEl exs a}

doFilterEl :: ToId a => [ExportSpec] -> [a] -> [a]
doFilterEl exs = filter (('elem' map esToName exs) . idToName . toId)
```

import and export lists.

## 4.1 Export and import lists

Export lists were handled first. The change in the code was done in the function that writes the interface file with the symbols defined by the module and their types, called `writeIface`. The interface that is the input of this function was filtered through another function, `filterEl`, which would keep only the entities in the export list, if it was available. The original source code is shown in Figure 4.1 and the new one in Figure 4.2. The code for function `filterEl` is shown in Figure 4.3.

This function is very simple. There are three types of exported entities that are affected by an export list: type synonyms, stored in list `synonyms`; type classes, stored in `classes`; and types identifiers, stored in `assumptions`. Each element of these lists

**Figure 4.4.** The original code of the function `readIface`.

```
readIface dir i
  = do
    let v = gen dir (importModule i)
        parseInterface v
```

**Figure 4.5.** The new code of the function `readIface`.

```
readIface dir i
  = do
    let v = gen dir (importModule i)
        filterI1 (importSpecs i) <$> parseInterface v
```

is compared with the export list if it is present. In this case, only the items that are present in the export list specification are returned by the function. If the export list is not present, all entities are returned.

The export lists were handled just before writing the interface file. Import lists are handled, analogously, just after reading the interface file. The results of parsing the interface file are filtered using the rules of the import list. The original code for function `readIface`, which read and parsed the interface file, is shown in Figure 4.4. The new code for this function is in Figure 4.5.

The parsed results are filtered using function `filterI1`. Its source code is shown in Figure 4.6. This function is a little bit more complicated because the import list can define the imported entities or the entities that should not be imported, when keyword `hiding` is used. Similarly to `filterE1`, it filters the three types of entities affected by import lists, checks if it is a `hiding` import or not, and removes the necessary elements.

## 4.2 Conflicting instances

Conflicting instances are instances of the same type class defined for the same type. Using such an instance is a compile time error in Haskell, since there is no reasonable way of deciding which instance to choose, in any such usage. Unfortunately, however, the prototype front-end performed no test to verify if conflicting instances existed. For implementing our proposal, we first modified the front-end to make this verification.

To implement this, a change had to be made in the type-checking algorithm to

**Figure 4.6.** The new code of the function `readIface`.

```

filterI1 :: Maybe (Bool, [ImportSpec]) -> Iface -> Iface
filterI1 Nothing i = i
filterI1
  (Just (hid, is))
  i@(Iface {synonyms = s, classes = c, assumps = a})
= i
  {synonyms = doFilterI1 hid is s,
   classes = doFilterI1 hid is c,
   assumps = doFilterI1 hid is a}

doFilterI1 :: ToId a => Bool -> [ImportSpec] -> [a] -> [a]
doFilterI1 hid is
= filter ((\x -> any (f x) $ map isToName is) . idToName . toId)
  where
  f :: Eq a => a -> a -> Bool
  f
    | hid = (/=)
    | otherwise = (==)

```

**Figure 4.7.** The original code of the instance check.

```

foldM (satstep phi p) [[]] delta

```

**Figure 4.8.** The new code of the instance check which does not allow more than one instance of the same class for the same type.

```

case delta of
  [delta_] -> satstep phi p [[]] delta_
  [] -> error "No instances found."
  _ -> error "More than one instance found."

```

give an error when more than one instance was found. The original code is shown in Figure 4.7, and the new code in Figure 4.8. The number of available instances is verified, and if no instances are found or if there are more than one instance found, error messages are reported.

**Figure 4.9.** Function `filterI1` on the intermediate version.

```

filterI1 :: Maybe (Bool, [ImportSpec]) -> Iface -> Iface
filterI1 Nothing i = i
filterI1
  (Just (hid, is))
  int@(Iface {synonyms = s, classes = c, assumps = a, instances = i})
= int
  {synonyms = doFilterI1 hid is s,
   classes = doFilterI1 hid is c,
   assumps = doFilterI1 hid is a,
   instances = filterInstancesI1 hid is i}

```

### 4.3 Intermediate version

To implement the intermediate version, the only change needed is in the `hiding` keyword of an import list. In function `filterI1`, instances need to be treated separately. Function `doFilterI1` cannot be used, because instances do not have a simple name like other entities: their identification is composed by the name of the type class and the name of the data type. Therefore, another function was created to deal with them, called `filterInstancesI1`. In the intermediate version, this function checks if an import list uses keyword `hiding` and, if so, it filters all instances that are in such import list. The new code for `filterI1` is shown in Figure 4.9. The code for `filterInstancesI1` is shown in Figure 4.10.

### 4.4 Final version

The final version requires changes not only in the handling of import lists, as the intermediate version, but also in the handling of export lists. Besides, changes in import lists do not only consider instances in import clauses with keyword `hiding`, but in all of them. To concentrate the treatment of keyword `hiding` in only one place, functions `doFilterI1` and `filterInstancesI1` were changed to receive a comparison function instead of a value indicating if this import clause has or not keyword `hiding`. This comparison function is the equality, `(=)`, when there is no `hiding` keyword, and the inequality, `(/=)`, when there is. The new code for function `filterI1` can be seen in Figure 4.11. The adaptation on `doFilterI1` and `filterInstancesI1` was simple, as can be seen in Figures 4.12 and 4.13.

**Figure 4.10.** Function `filterInstancesI1` on the intermediate version.

```

filterInstancesI1 :: Bool -> [ImportSpec] -> [Inst] -> [Inst]
filterInstancesI1 False is = id
filterInstancesI1 True is
  = filter
    ((\x -> any ((/=) x) $ mapMaybe importSpecToInstanceSpec is)
     . instToInstanceSpec)

importSpecToInstanceSpec :: ImportSpec -> Maybe (Id, Id)
importSpecToInstanceSpec (IInstance n m) = Just (toId n, toId m)
importSpecToInstanceSpec _ = Nothing

instToInstanceSpec :: Inst -> (Id, Id)
instToInstanceSpec (Inst c [t] _) = (c, toId t)
instToInstanceSpec _ = error "instToInstanceSpec _"

```

**Figure 4.11.** Function `filterI1` on the final version.

```

filterI1 :: Maybe (Bool, [ImportSpec]) -> Iface -> Iface
filterI1 Nothing i = i
filterI1
  (Just (hid, is))
  int@(Iface {synonyms = s, classes = c, assumps = a, instances = i})
= int
  {synonyms = doFilterI1 op is s,
   classes = doFilterI1 op is c,
   assumps = doFilterI1 op is a,
   instances = filterInstancesI1 op is i}
where
  op :: Eq a => a -> a -> Bool
  op
    | hid = (/=)
    | otherwise = (==)

```



**Figure 4.12.** Function `doFilterI1` on the final version.

```
doFilterI1
  :: ToId a => (Name -> Name -> Bool) -> [ImportSpec] -> [a] -> [a]
doFilterI1 op is
  = filter ((\x -> any (op x) $ mapMaybe isToName is) . idToName . toId)
```

**Figure 4.13.** Function `filterInstancesI1` on the final version.

```
filterInstancesI1
  :: ((Id, Id) -> (Id, Id) -> Bool) -> [ImportSpec] -> [Inst] -> [Inst]
filterInstancesI1 op is
  = filter
    ((\x -> any (op x) $ mapMaybe importSpecToInstanceSpec is)
     . instToInstanceSpec)
```

**Figure 4.14.** Function `filterEl` filtering instances.

```
filterEl :: Module -> Iface -> Iface
filterEl (Module _ _ _ _ Nothing _ _) i = i
filterEl
  (Module _ _ _ _ (Just exps) _ _)
  int@(Iface {synonyms = s, classes = c, assumps = a, instances = i})
  = int
  {synonyms = doFilterEl exps s,
   classes = doFilterEl exps c,
   assumps = doFilterEl exps a,
   instances = filterInstancesEl exps i}
```

The implementation of filtering export lists is similar, but simpler because there is no need to consider cases of keyword `hiding`. Function `filterEl` was changed to handle instances also, as done with import lists. The new code for this function is shown in Figure 4.14. As with import lists, a separate function filters instances, in this case called `filterInstancesEl`. The code for this function is similar to `filterInstancesI1` and is shown in Figure 4.15.

**Figure 4.15.** Function `filterInstancesEl`.

```

filterInstancesEl :: [ExportSpec] -> [Inst] -> [Inst]
filterInstancesEl exps
  = filter
    $ ('elem' mapMaybe exportSpecToInstanceSpec exps)
      . instToInstanceSpec

exportSpecToInstanceSpec :: ExportSpec -> Maybe (Id, Id)
exportSpecToInstanceSpec (EInstance n m) = Just (toId n, toId m)
exportSpecToInstanceSpec _ = Nothing

```

## 4.5 Testing

In order to test the implementation, comprehensive test cases were elaborated<sup>2</sup>, with two separated tests for import lists, one with the intermediate version and one with the final. For each of them, 16 cases were created. This number of tests, 16, is a consequence of the fact that there are 2 imports in each test, each one importing a module which defines a conflicting instance, and, for each import, there are 4 ways to import it: (a) without import lists, (b) with an empty import list, (c) with the import list specifying an imported instance and (d) with the import list hiding an instance. The file name of each test corresponds to the expected result. If the file name is prefixed with `Overlap`, it is expected that the compiler will report an error of overlapping instances. If the file name is prefixed with `NoInstance`, it is expected that the compiler will not find an instance to be used. If the file name is prefixed with `Ok`, there are no errors. The suffix of each file is the number of the test. A script was created that compiles each of these test cases and checks if the output of the compiler is in accordance with the prefix of the file name.

To test the export list filter, nine cases were considered, because there are only three types of exportation: (a) no export list, (b) empty export list and (c) an instance in the export list. The tests for the export list were done in the same way as for import lists, and the same script was used. For all tests, every result was correct.

---

<sup>2</sup>The test cases are available at <https://github.com/marcotmarcot/scoped-instances-tests>

## 4.6 Conclusion

The implementation is very simple in the prototype, and should not be complicated in a production compiler, if done by a specialist in the compiler. As the testing coverage is complete, we have confidence that the proposal achieves its goals of controlling the scope of instances.



## Chapter 5

# Extending Haskell's Module System Formal specification

The module system of Haskell 98 has been formally specified [Diatchki et al., 2002] without dealing with type class instances. This chapter presents an extension of this formalization for dealing with type class instances, including the changes needed in this formal specification in order to cope with both the intermediate and final alternatives of our proposal. The work in which the formalization is made does not provide the complete code of the formalization, but the code is available on the web<sup>1</sup>.

The code models `Name` as a wrapper around a `String`, and it is stated in the work that type class instances were not considered because it is not possible to refer to them by a name [Diatchki et al., 2002, Section 3.1]. We propose that names of instances be written as they occur in export and import clauses (as presented in Figures 3.1 and 3.2). By doing this, there is no need to change data type `Name`, nor data type `Entity` used for describing exported and imported entities.

---

<sup>1</sup><http://yav.purely-functional.net/publications/modules98-src-21-Nov-2005.tar.gz>.

**Figure 5.1.** 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
```

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 5.1, is used to distinguish type class instances from other entities. Function `isInst` is used in the same manner as function `isCon`, defined in the paper [Diatchki et al., 2002, Section 3.1]. Another auxiliary function that should be defined is a filter for type class instances, called, say, `instances`, as in Figure 5.1, to be used for the changes introduced in our extension of the formalization.

## 5.1 Haskell and the intermediate alternative

Our proposal can be applied to both Haskell 98 and 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 syntactic 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` [Diatchki et al., 2002, Section 5.2] needs to be changed. The old version of the function is presented in Figure 5.2 and the new version in Figure 5.3. The difference between them is just that, when an export list is available, as in the `Just es` case, the instances are exported with what is specified in the export list. The instances, then, are always exported, as defined in Haskell 2010 report [Marlow, 2010, Section 5.4].

The other change needed, which is related to imported entities, is in 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 5.4 and 5.5. 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 in the hiding

**Figure 5.2.** Function exports as in [Diatchki et al., 2002, 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

```

**Figure 5.3.** 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

```

**Figure 5.4.** The function `incoming` as it is on [Diatchki et al., 2002, Section 5.3], for reference.

```

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

```

**Figure 5.5.** The new `incoming` function that also deals with instances.

```

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

```

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 in the hiding list, it will be imported, because it is included in `exps`.

## 5.2 The final alternative

To specify the final alternative, the consideration about how to use instances by 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 handled in the same way as other Haskell entities, so that the specification that worked for them works also for instances.



# Chapter 6

## Related work

The work of Named instances [Kahl and Scheffczyk, 2001] solves issues related to those discussed in our work. In that work a new name must be given for each instance, and the name must be used to reference the defined instance. This constitutes a very significant change to the language. Our proposal is simpler, since it requires fewer changes to 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 of *scoped instances* [Dijkstra et al., 2007], which suggests a language extension for Haskell that allows instances to be defined inside `let` clauses. An example is given in Figure 6.1. 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” [Dreyer et al., 2007]. Their work also identifies drawbacks of the current state of the Haskell’s

**Figure 6.1.** Example of scoped instance extracted from [Dijkstra et al., 2007, Section 6].

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

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 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.

Type classes hold a strong correlation with Scala implicits [Bruno C. d. S. Oliveira, 2010], in which a type class declaration in Scala is done by using a trait, and a type class instance can be obtained by creating an object which implements the trait. To avoid having to pass the object (which would be the instance in Haskell) as a parameter in each call of the function, this object can be declared implicit, in which case it is passed implicitly to functions that expect an object of this type. Even though most of the times the name of this kind of object is not used, all objects of this type must have a name, and this name can be imported and exported. The decision of which implicit to use can be done by passing the name of the object as an explicit parameter, so that, even in a single module, more than one implicit for the same type can be used.

# Chapter 7

## Conclusion

An article presenting this proposal was published in the 2011 edition of the Brazilian Symposium on Programming Languages [Gontijo and Camarão, 2011].

The Haskell language extension proposed in this work is relatively simple and gives more freedom to programmers. On the negative side, it 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.

### 7.1 Contributions

On the positive side, our proposal makes only small changes to the language syntax and semantics. It gives more control to programmers which may now construct 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. This is demonstrated by the fact that the formalization does not need to be changed to deal with instances in our final proposal.

As far as we can see, our proposal does not interfere with other Haskell language feature, mechanism or known extensions.

## 7.2 Future work

This work has presented both syntactic and semantic details of our proposal. Both syntax alternatives have been implemented in a prototype. An implementation 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 opportunity to use the extension in production code, enabling an evaluation of the utility of the extension in the real world.

# Bibliography

- Arnold, K., Gosling, J., and Holmes, D. C. (2006). *The Java Programming Language*. Prentice Hall, 4rd edition.
- Bruno C. d. S. Oliveira, Adriaan Moors, M. O. (2010). Type classes as objects and implicits. In *OOPSLA*, pages 341–360.
- Diatchki, I., Jones, M., and Hallgren, T. (2002). A formal specification of the Haskell 98 module system. In *Proc. of the 2002 Haskell Workshop*.
- Dijkstra, A. et al. (2007). Modelling Scoped Instances with Constraint Handling Rules. <http://www.cs.uu.nl/wiki/bin/viewfile/Ehc/ModellingScopedInstancesWithConstraintHandlingRules?rev=1.1;filename=20070406-2213-icfp07-chr-locinst.pdf>.
- Dreyer, D. et al. (2007). Modular Type Classes. In *SIGPLAN Notices*, volume 42, pages 63–70.
- Faxén, K. (2002). A static semantics for Haskell. *Journal of Functional Programming*, 12:295--357.
- Gontijo, M. and Camarão, C. (2011). Controlling the scope of instances in Haskell. In *Proceedings of the 2011 Brazilian Symposium on Programming Languages*, São Paulo. Brazilian Computer Society.
- Hall, C. V. et al. (1996). Type classes in Haskell. In *ACM Transactions on Programming Languages and Systems*, volume 18, pages 109–138.
- Hudak, P. et al. (2007). A history of Haskell: Being lazy with class. In *HOPL-III: Proc. 3rd ACM SIGPLAN Conf. History of Programming Languages*, pages 1–55, San Diego, CA, USA. ACM Press.
- Kahl, W. and Scheffczyk, J. (2001). Named Instances for Haskell Type Class. Technical report UU-CS-2001-62, Universiteit Utrecht.

- M. Jones (1994). *Qualified Types: Theory and Practice*. PhD thesis, Distinguished Dissertations in Computer Science. Cambridge Univ. Press.
- Markstrum, S. (2010). 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*.
- Marlow, S., editor (2010). *Haskell 2010: Language Report*. <http://www.haskell.org/onlinereport/haskell2010/>.
- Milner, R., Tofte, M., and Harper, R. (1988). The Definition of Standards ML, version 2. Technical report ECS-LFCS-88-62, Edinburgh University, Computer Science Dept.
- Odersky, M., Spoon, L., and Venners, B. (2012). *Programming in Scala*. Artima, 2nd edition.
- Peyton Jones, S., Jones, M., and Meijer, E. (1997). Type classes: An exploration of the design space. In *Haskell Workshop*.
- Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press.
- Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley, 3rd edition.
- Wadler, P. and Blott, S. (1989). How to make ad-hoc polymorphism less ad hoc. In *Proc of the 16th ACM Symposium on Principles of Programming Languages*, pages 60–76.