Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

František Farka

Maintainable type classes for Haskell

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	RNDr. Petr Pudlák, Ph.D.
Study programme:	Informatics
Specialization:	Theoretical Computer Science

Prague 2014

I would like to thank my supervisor for his guidance and immense patience as I was not always the most diligent student. I would also like thank my friends for their valuable comments and insight the helped me to gain in many discussion on the topic of this thesis.

And last but not least I owe my deepest gratitude to my parents for their unlimited love and support.

Název práce: Udržovatelné typové třídy v Haskellu

Autor: František Farka

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Petr Pudlák, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: V této práci se zaměřujeme na dlouhodobý problém v systému typových tříd jazyka Haskell. Konkrétně se zabýváme možnostmi zpětně kompatibilních úprav v existujících hierarchiích tříd. V první části práce podáváme stručný přehled jazyka. Následující část shrnuje stávající navrhovaná řešení problému a rozebírá jejich vlastnosti. Na základě tohoto rozboru předkládáme náš vlastní návrh na jazykové rozšíření.

V předposlední části uvádíme několik možných užití jazykového rozšíření a srovnáváme jej s ostatními řešeními. Součástí práce je také proof-of-concept implementace rozšíření pro kompilátor GHC, která je stručně popsána v poslední části.

Klíčová slova: Haskell, typové třídy, udržovatelnost, výchozí instance nadtříd

Title: Maintainable type classes for Haskell

Author: František Farka

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Petr Pudlák, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract:

In this thesis we address a long-term maintainability problem in Haskell type class system. In particular we study a possibility of backward-compatible changes in existing class hierarchies. In the first part of the thesis we give a brief overview of the language. The following part summarizes current proposed solutions to the problem and analyzes their properties. Based on this analysis we derive our own language extension proposal.

In the penultimate chapter we present several possible applications of the language extension and compare the extension to other solutions. As a part of the thesis we also give a proof-of-concept implementation of the extension for the GHC compiler, which is briefly described in the last part of this thesis.

Keywords: Haskell, type classes, maintainability, default superclass instances

Contents

Introduction **4**
- Motivation . . . 4
- Structure of the thesis . . . 5

1 The Haskell Language Description **6**
- 1.1 Program Structure . . . 6
- 1.2 Lexical Structure . . . 7
  - 1.2.1 Comments . . . 7
  - 1.2.2 Identifiers and Operators . . . 8
  - 1.2.3 Literals . . . 8
  - 1.2.4 Layout . . . 9
- 1.3 Expressions . . . 9
  - 1.3.1 Error Handling . . . 9
  - 1.3.2 Variables, Constructors and Operators . . . 9
  - 1.3.3 Function Application and Lambda Abstraction . . . 9
  - 1.3.4 Conditionals . . . 10
  - 1.3.5 Let Expressions . . . 10
  - 1.3.6 Case Expressions . . . 10
  - 1.3.7 Do Notation . . . 10
  - 1.3.8 Pattern Matching . . . 11
- 1.4 Declarations and Bindings . . . 11
  - 1.4.1 Kinds . . . 11
  - 1.4.2 User-defined Data Types . . . 12
  - 1.4.3 Type Classes . . . 12
  - 1.4.4 Nested Declarations . . . 14
  - 1.4.5 Function and Pattern Bindings . . . 14
- 1.5 Modules . . . 15
- 1.6 Predefined Types and Classes . . . 16
  - 1.6.1 Data Types . . . 17
  - 1.6.2 Classes . . . 17
- 1.7 Compiler Pragmas . . . 19
- 1.8 Static Semantics . . . 20

2 Maintainability Problem **26**
- 2.1 Altering Type Class Hierarchy . . . 26
- 2.2 Currently Possible Solutions . . . 28
  - 2.2.1 Functor – Applicative – Monad . . . 28
  - 2.2.2 Subclass to Superclass Instance . . . 28

2.2.3 The Strathclyde Haskell Enhancement 29
2.3 Design Goals of Language Extension 29

3 Previous Proposals 31
3.1 Default Superclass Instances by GHC 31
3.1.1 Our Conclusion 32
3.2 Class Alias Proposal by John Meacham 33
3.2.1 Our Conclusion 34
3.3 Superclass Defaults 34
3.3.1 Our Conclusion 35
3.4 Class System Extension Proposal 36
3.4.1 Conclusion 37

4 Language Extension Proposal 38
4.1 Language Extension Features 38
4.1.1 Default Instances 38
4.1.2 Default Method Implementation 40
4.1.3 Class Aliases 41
4.2 Our Proposal 43
4.3 Superclass Default Instances 45
4.3.1 Syntax of the Extension 45
4.3.2 Semantics of the Extension 47
4.4 Class Aliases 50
4.4.1 Syntax 50
4.4.2 Relation to The Superclass Default Instances 50

5 Relation to the Language Platform 52
5.1 Relation to the Existing Language Extensions 52
5.1.1 Multi-parameter Type Classes 52
5.1.2 Default Method Signatures 53
5.1.3 Functional Dependencies 53
5.1.4 Flexible Instances and Undecidable Instances 53
5.2 Comparison of Current Solutions to Default Superclass Instances . 53
5.2.1 Backward Incompatible Change in Hierarchy 53
5.2.2 Subclass to Superclass Instance 54
5.2.3 SHE 54
5.3 Applications of Proposed Extension 54
5.3.1 Bind and Pointed 54
5.3.2 Standard Numeric Classes 55
5.3.3 Traversable 56

6 Implementation 57
6.1 Compiler Architecture 57
6.1.1 The Haskell Compiler 58
6.2 Changes to the Compiler 60
6.2.1 Extension Flag 60
6.2.2 Parser 60
6.2.3 Renamer 61
6.2.4 Type Checker 61

6.2.5 Desugaring 62
6.3 Concluding notes 62

Conclusion **63**

Further work **64**

References **67**

A Lexical Structure **69**

B Syntax of the Language **72**

C Debugging Outputs **73**
C.1 Parser Output 73
C.2 Renamer Output 74
C.3 Type Checker Output 74

Introduction

Motivation

Haskell is a purely functional programming language for general-purpose applications. Haskell originated in the academic sector [17], thus the authors had—and anyone who is working on the Haskell platform still has—an unique opportunity to incorporate new and innovative features into the language [22]. The language truly availed of the situation and programmers may now use to their advantage e. g. higher order functions, non-strict semantics, algebraic data types, or static polymorphic typing.

Besides the role the language played in the academic community for last two decades it also found its way into the industry. Some of the biggest software companies of the world of today have successfully made Haskell part of their technological stack. We may illustrate this advancement on the examples of companies such as Google [31] and Facebook [24]. The position of the Haskell language platform is also endorsed by professional development tools and environments that has emerged, e. g., development environment and business analysis platform [5].

However, regardless the question whether it is in spite of the years of evolution in academic community or whether it is a result of its origins in this community, the Haskell language suffers from some issues that that hold back the evolution of libraries [37] and bring problems into its day-to-day use. In this thesis we focus on a particular instance of such problem – the design of type classes and long term maintainability of type class hierarchies.

The Haskell type class system uses classes as a description of methods available on types belonging to the class and instances as a mean of providing implementation of the methods for certain data type. Current design requires an instance of class and data type to exist when there is an instance of any child class in the hierarchy and the given data type. In a situation the classes are provided by a library and instantiated on a custom data types by user of the library it makes if difficult for the author of the library to introduce new parents to existing classes. Such a change in a hierarchy is not backward compatible—it requires instances that are not guaranteed to exist—and may case the source code not to compile with a new version of the library.

The problem was identified back in 2006 by Jón Fairbairn[10]. In this thesis we discuss several solutions and language extension proposals that attempt to address the problem under our consideration. Despite the number of proposals and the amount of time since Fairbairn's realisation there is no general solution to this problem yet. There is being deployed – contemporary to the writing of this thesis – a fix to one particular instance of this problem, which we discuss in section 2.2.1, by The Glorious Glasgow Haskell Compilation System (GHC). This fix demonstrates some practical difficulties that the lack of general mechanism causes.

In this thesis we attempt to analyse the current situation and, based on this analysis, design a language extension that provides a general mechanism for solving an above stated problem. We also give a proof-of-concept implementation of our extension in GHC.

Structure of the Thesis

The thesis is logically divided into several chapters. In the first chapter we bring an overview of Haskell language. We deem this overview necessary as a brief reference for further discussion on the class hierarchy problem and for proper understanding of source code examples included in this thesis.

In the second chapter we give a general description of the problem and present existing approaches and possible solutions. In the third chapter we present existing language proposals and shortly discuss every proposal.

The fourth chapter consists of our own language extension proposal that addresses conclusion made on previous existing proposals in the Chapter 4. We dedicate the fifth chapter to the connection of our extension with existing language framework. We comment on the relation to other language extensions and compare our approach to the possibilities listed in the Chapter 2.

In the last chapter we give a short description of the implementation of our extension. In the end of the thesis we provide a conclusion and list further work objectives related to this work.

Note on Notation Throughout this thesis we use particular typographical variants of standard Haskell operators. We believe this approach makes source code examples more understandable. Following Haskell operators and functions

```
\ undefined forall >>= >> == ++
```

are typeset respectively:

λ $\perp$ $\forall$ $\ggg$ $\gg$ $\equiv$ $+\!\!+$

Other typographical conventions are described later when appropriate symbol is first defined.

Chapter 1

The Haskell Language Description

Haskell is a purely functional programming language. Haskell provides e. g. higher order functions, non-strict semantics, static polymorphic typing, and user defined algebraic data types.

In this chapter we give a short description of the programming language. We do not intend to cover the language in all its details but rather present only a summary that is necessary for understanding the examples throughout the thesis. We further elaborate only on the parts that are vital to the problem under consideration, i. e., mostly the class system and instances. We omit any description of the standard library in the form of the Prelude, or any libraries.

We use the The Haskell 2010 Language Report [22] (further simply referred to only as The Report) as our primary source. The structure of this chapter also mostly respects the structure given in The Report.

However, The Report gives only an informal semantics of the language. A formal static semantics is presented by Hall et al. in [14], Jones and Wadler in [19], later by Faxén in [11]. Some insight is also brought by Sulzmann and Wang in [33]. We describe the most of the semantics only informally with the exception of classes and instances in the Section 1.8. We consider the formal specification in this case to be vital for further design of an extension and its implementation.

Where the technical description given in our thesis or The Report is not sufficient for proper understanding of some concepts, we refer the reader to any book on the Haskell language, e. g., the Real World Haskell [29] by O'Sullivan et al.

1.1 Program Structure

The Haskell program is hierarchically structured in the terms of the following constructs:

Modules are the topmost structures in a Haskell program. Formally, the program is a set of *modules* with one distinctive module *Main*.

Declarations Every module is a set of *declarations*, described in the Section 1.4.

Expressions are used to form declarations. An expression denotes a *value* with a *static type*.

Lexical structure captures a mean of representing the program and the aforementioned syntactic structures.

Values and types are separated. The type system permits *parametric polymorphism* using a Hindley-Milner type system by Hindley [16], Milner [28] and Damas [6]. The *ad-hoc* polymorphism is possible in the form of *overloading* with the *type classes*. We discuss the ad-hoc polymorphism in the Section 1.8.

Errors are semantically equivalent to $\perp$ (bottom). They cannot be technically distinguished from nontermination.

There are six kinds of names used in Haskell programs:

- *variables* and *constructors* denoting values,
- *type variables*, *type constructor*, and *type classes* that related to the type system, and
- *module names* identifying modules.

The identifiers reserved for variables or type variables begin with a lowercase letter or an underscore, the other four types of identifiers begin with an uppercase letter. An identifier cannot be used both as a type constructor and a class in the same scope. Otherwise, an identifier may simultaneously denote multiple entities, e. g., a module and a class.

1.2 Lexical Structure

In this section we focus only on several key lexical elements of the language. The formal grammar is attached in the Appendix A.

1.2.1 Comments

The Haskell language provides two variants of comments. There are single line comments that begin with two or more consecutive dashes and extended to the following newline, e. g.:

```
-- This is a one line comment
```

There are also multiple-line comments that begins with "{-" and are terminated by "-}". These comments can be nested in arbitrary manner. Thus this is a valid comment:

```
{-
        This is a comment

        {-
                This is still a comment
        -}

-}
```

Both types of comments are considered white space from the lexical point of view. The second type of comments is also used for compiler pragmas, which are described in the Section 1.7.

1.2.2 Identifiers and Operators

A Haskell identifier consists of a letter or an underscore followed by a sequence of letters, digits, underscores, and single quotes. Identifiers are case sensitive. Identifiers divide into two namespaces depending on the first character. Identifiers beginning with a lowercase letter or an underscore are used for variables and type variables. Identifiers beginning with an uppercase letter are used for constructors, type constructors, classes, and modules. Following identifiers are reserved:

```
case class data default deriving do else
foreign if import in infix infixl
infixr instance let module newtype of
then type where _
```

Operator is a sequence of one or more special characters. Depending on first character operators divide in two categories. In the case the first character is the colon, the operator is a constructor. Otherwise, operator is an ordinary identifier. Following operators are reserved:

```
.. : :: = \ <- -> @ ~ =>
```

In the following text we prefer following typographical variants of these operators respectively:

`..` `:` $::$ `=` λ $\leftarrow$ $\rightarrow$ `@` `~` $\Rightarrow$

All operators are infix. Operators and identifiers may be qualified, i. e., prepended with a module name.

1.2.3 Literals

The Haskell language allows several type of literals.

Integers are written either as ordinary decimals, in octal notation prefixed with "`0o`" or "`0O`", or in hexadecimal notation prefixed with "`0x`" or "`0X`".

Floating point literals are always decimals in either decimal point notation or in scientific notation using "`E`" or "`e`" and mantissa, e. g.:

```
3.145   217e-2  1618E-3
```

Character literals are denoted by single quote (`'`)

Strings are denoted by quotes (`"`). Escaping with backslash is possible. A string literal is an abbreviations for list of characters.

1.2.4 Layout

The language allows semicolon and brace-free style of notation. In this case the indentation is used instead. The Report [22] specifies the proper layout rules. Informally, blocks are considered to be the source lines with the same level of indentation. We give a short example of a function defined using the layout mechanism and then the same function without the layout.

```
foo :: a → b → (b, a)
foo a b = swap makePair a b where
        makePair a b = (a, b)
        swap f a b = f b a
```

The same function desugared of layout:

```
;foo :: a → b → (a, b)
;foo a b = swap makePair a b where
        {makePair a b = (a, b)
        ;swap f a b = f b a
}
```

In the second case the indentation does not convey any information.

1.3 Expressions

In this section, we describe the syntax and the semantics of the Haskell language with respect to the problem under our consideration.

1.3.1 Error Handling

Errors during expression evaluation are denoted by ⊥. An error and any program nontermination are indistinguishable. Haskell specifies two functions that directly cause an error in the evaluation:

```
error :: String → a
⊥ :: a
```

A call to either of the functions terminates execution. Actual error message is dependent on a particular implementation.

1.3.2 Variables, Constructors and Operators

Identifiers and operators can be used both in prefix and infix notation and can be partially applied. It is possible to apply identifiers infix by enclosing the operator with backtics, i. e. `a ‘function‘ b`. Operators can be used and partially applied in prefix notation using *section*, i. e., notation `(op) a b` for prefix application and notations `(a op) b` and `(op b) a` for partial application in first and second operand respectively.

1.3.3 Function Application and Lambda Abstraction

Function application is written as $\texttt{exp}_1$ $\texttt{exp}_2$ and associates to the left, thus parenthesis may be omitted. This also allows partial application.

Lambda abstractions are written as λ `pat`$_1$ `...pat`$_n$ `-> exp` where `pat`$_1$ to `pat`$_n$ are patterns. If the pattern fails to match, then the result is $\perp$.

Operators are infix applied with respect to the sections.

1.3.4 Conditionals

Haskell provides `if` conditional expression as many programming languages do, with the exception that the else-clause is mandatory:

```
        if exp₁ [;] then exp₂ [;] else exp₃
```

The first expression `exp`$_1$ is of the Boolean type. The other two expressions have the same type which is also the type of the whole conditional statement. Otherwise conditional statement results in an error. Semicolons are optional.

1.3.5 Let Expressions

Let expressions introduce nested, lexically scoped, mutually recursive set of declarations of the general form:

```
        let { d₁, ..., dₙ } in exp
```

Let expression is lazily evaluated, thus any error in declarations does not occur until the expression is evaluated.

1.3.6 Case Expressions

Case expressions allow conditional evaluation among several alternatives. The general form is:

```
case exp of
        pat₁ [ |guard₁,₁, ..., guard₁,ₙ] → exp₁ [ where decls₁ ]
        ...
        patₙ [ |guardₙ,ₙ, ..., guardₙ,ₙ] → expₙ [ where declsₙ ]
```

The *guards* and where clauses are optional. In the case the `where` clause is present, it contains a set of ordinary declarations. The guards have the general form of:

```
        exp             -- boolean guard
        pat ← exp       -- pattern guard
        let decls       -- local bindings
```

A case expression is evaluated by matching patterns *pat*$_1$ to *pat*$_n$ sequentially. If a pattern matches, then the guarded expression is evaluated. Boolean guard succeeds if the expression *exp* evaluates to *True*. Pattern guards succeeds if the expression *exp* matches the pattern *pat*. Local bindings always succeed and introduce names defined in *decls*. If no alternative succeeds then, the case expression evaluates to $\perp$.

1.3.7 Do Notation

Do notation is only a more convenient syntax for monadic operations described in the Section 1.6.2. We consider the following examples provided in The Report to be descriptive enough. Assuming source code using `Monad` class operations $\gg=$ and $\gg$:

```
putStr "x: "     ≫
getLine          ≫= λl →
return (words l)
```

following do notation is equivalent:

```
do      putStr "x: "
        l ← getLine
        return (words l)
```

Note that this transformation can by carried as a source code transformation.

1.3.8 Pattern Matching

Pattern matching is used as a part of other language constructs. Pattern consists of an expression matched against value. The match occurs on the structure of the expression and the value, and results in three cases: it either fails, succeeds, or diverges, i. e., formally results in $\perp$. In case the match succeeds the variables of the pattern are bind to the values accordingly.

Patterns occur in two variants, as *refutable patterns* and as *irrefutable patterns* denoted by `~(...)`. Matching of refutable patterns is strict and matching against $\perp$ diverges. Matching irrefutable patterns is lazy and always succeeds. In later case variable bindings are resolved when the irrefutable pattern is evaluated.

1.4 Declarations and Bindings

In this section we describe the syntax and the semantics of language declarations. The declarations are divided into three categories. We describe user-defined data types, type classes and related declarations, and nested declarations. Separately from these three we describe the *kind* mechanism.

1.4.1 Kinds

Kinds ensure the validity of type expressions. Kinds are constructed in the following way:

$$\begin{array}{lll} * & \text{is a kind} & (1) \\ \kappa_1 \to \kappa_2 & \text{is a kind} \iff \kappa_1 \text{ and } \kappa_2 \text{ are kinds} & (2) \end{array}$$

The first case (1) is the kind of nullary type constructors, e. g., built-in types `Integer` and `Char` or user-defined data types without type variables. The other case (2) is the composite kind of types that take a type of the kind κ_1 and return a type of the kind κ_2. Normally, kinds are entirely implicit and not visible to the programmer. However there are language extension which allow direct access to the kind system, e. g. Kind polymorphism extension [13].

The exact kind inference mechanism is described in The Report [22].

1.4.2 User-defined Data Types

User-defined data types occur in three variants: algebraic data types as `data` declarations, renamed data types as a `newtype`, and type synonyms as `type` declarations.

An algebraic data type has the form

`data` *cx* $\Rightarrow T\ u_1\ \ldots\ u_k = K_1\ t_{1,1}\ \ldots\ t_{1,k_1}\ |\ldots\ |K_n\ t_{n,1}\ \ldots\ t_{n,k_n}$

where *cx* is a context and K_1 to K_n are new *data constructors*. The declaration introduces a new type with the *type constructor* T. The type has a kind $\kappa_1 \rightarrow \ldots \rightarrow \kappa_k \rightarrow *$ where κ_1 to κ_k are kinds of the type variables u_1 to u_k respectively.

Optionally, it is possible to use labels for the fields of a data type. Then the syntax of the data type is

```
data Foo a = Foo { bar :: a, baz :: a }
```

and two field selectors are brought into the external scope of the data type:

```
bar :: Foo a → a
baz :: Foo a → a
```

The field identifiers must be unique within the scope. A data type with labeled fields is called a *record*. It is still possible to use the constructor of a record as an ordinary data constructor and the fields will be associated positionally.

The `type` synonym has a form

`type` $T\ u_1\ \ldots\ u_k$ `=t`

and introduces a new type with a constructor T. The new type $(Tt_1 \ldots t_k)$ is equivalent to the type $t[t_1/u_1, \ldots, t_k/u_k]$ where type variables are substituted. The type synonym constructor must be fully applied. The type synonym is a syntactic construct and is interchangeable for the appropriate type everywhere except for instance declaration.

A declaration of the form

`newtype` $T\ u_1\ \ldots\ u_k$ `=N t`

brings a new data type renaming into the scope. A `newtype` construct creates a new type, with type constructor N, which is different from original type, unlike the type synonyms. The type must be constructed with the type constructor and may be pattern matched upon.

1.4.3 Type Classes

A *type class* declaration introduces a new class and the operations on the type associated with the class through an *instance*. The declaration must occur as a top level declaration with the syntax as shown in the Figure 1.1.

Thus in general the class declaration has the form

`class` *cx* $\Rightarrow C\ u$ `where` *cdecls*

where the context *cx* specifies the *superclasses* of C, u is the type variable of the type which is an instance of C, and *cdecls* are class declarations. Superclass relation must be acyclic. The declarations in *cdecls* part are:

topdecl	→	`class` [*scontext* `=>`] *tycls tyvar* [`where` *cdecls*]	
scontext	→	*simpleclass*	
	|	`(` $simpleclass_1$ `,` … `,` $simpleclass_n$ `)`	$(n \geq 0)$
simpleclass	→	*qtycls tyvar*	
cdecls	→	`{` $cdecl_1$ `;` … `;` $cdecl_n$ `}`	$(n \geq 0)$
cdecl	→	*gendecl*	
	|	(*funlhs* | *var*) *rhs*	

Figure 1.1: Class declaration syntax

topdecl	→	`instance` [*scontext* `=>`] *qtycls inst* [`where` *idecls*]	
scontext	→	*simpleclass*	
	|	`(` $simpleclass_1$ `,` … `,` $simpleclass_n$ `)`	$(n \geq 0)$
simpleclass	→	*qtycls tyvar*	
inst	→	*gtycon*	
	|	`(` *gtycon* $tyvar_1$ … $tyvar_k$ `)`	$(k \geq 0,$ tyvars distinct$)$
	|	`(` $tyvar_1$ `,` … `,` $tyvar_k$ `)`	$(k \geq 2,$ tyvars distinct$)$
	|	`[` *tyvar* `]`	
	|	`(` $tyvar_1$ `->` $tyvar_2$ `)`	
idecls	→	`{` $idecl_1$ `;` … `;` $idecl_n$ `}`	$(n \geq 0)$
idecl	→	(*funlhs* | *var*) *rhs*	
	|		(empty)

Figure 1.2: Instance declaration syntax

- A new method declaration that is visible in the external scope of the class. The method name must be unique within the scope.
- A fixity declaration of a class method.
- A *default class method* implementation for any of the class methods. This implementation is used when no binding is given in an instance declaration.

Instances of a particular class and a given type are introduced by syntactic construct described in the Figure 1.2.

Assume the above mentioned class C. Then the general corresponding instance is

`instance` $cx' \Rightarrow C$ `(`$T\ u_1 \ldots u_k$`)` `where` *idecls*

where the type $(Tu_1 \ldots u_k)$ must by a type constructor applied to type variables and it must not be a type synonym. The type variables u_1 to u_k must be all different.

gendecl	→	*vars* `::` [*context* `=>`] *type*	(type signature)
	|	*fixity* [*integer*] *ops*	(fixity declaration)
	|		(empty declaration)
ops	→	op_1 `,` ... `,` op_n	($n \geq 1$)
op	→	*varop* | *conop*	
vars	→	var_1 `,` ... `,` var_n	($n \geq 1$)
fixity	→	`infixl` | `infixr` | `infix`	

Figure 1.3: Nested declarations syntax, part 1

The declaration may contain bindings for the methods of the class C. Some or all of the bindings may be omitted – in such case the default class method is used, if present. Otherwise, computation results in $\bot$.

Multiple instances of the same type and class are prohibited. The type and class of the instance must also have the same kind as the instantiated type. The instance must also satisfy context constraints implied by the superclasses of the corresponding class.

1.4.4 Nested Declarations

There are three more language constructs – type signatures, fixity declarations, and function and pattern bindings. The syntax of these constructs is described in figures 1.3 and 1.4.

All three constructs may be used both in the top level of the module and as a nested declaration.

Type signatures specify types of variables that may refer to the context. One signature may specify the type of multiple variables. Every variable in the type signature must have a binding in the scope of the type signature and every signature must be uniquely given; multiple signature declarations for one variable are invalid. If the type signature is provided, then the variable is treated as having a principal type.

A fixity declaration gives the precedence and associativity of one or more operators. The integer in declaration must be in the range 0 to 9 where 0 binds the least and 9 the most tight. The fixity keywords `infix`, `infixl`, and `infixr` have the meaning left-associative, right-associative, and non-associative binding respectively. The operators in operator list *ops* may be both variables and data type constructors.

1.4.5 Function and Pattern Bindings

The static semantics of function and pattern bindings is specified by The Report [22] as an application of Hindley-Milner type inference. However, there is a difference in generalization of type variables and Haskell is more restrictive in the generalization step. This is called the *monomorphism restriction*.

A binding b_1 is called dependent on a binding b_2, if b_1 contains a free identifier

decl	→	(*funlhs* | *pat*) *rhs*	(function or pattern binding)
funlhs	→	*var apat* `{` *apat* `}`	
	|	*pat varop pat*	
	|	(*funlhs*) *apat* `{` *apat* `}`	
rhs	→	`=` *exp* [`where` *decls*]	
	|	*gdrhs* [`where` *decls*]	
gdrhs	→	*guards* `=` *exp* [*gdrhs*]	
guards	→	`|` $guard_1$, ... , $guard_n$	$(n \geq 1)$
guard	→	*pat* ← *infixexp*	(pattern guard)
	|	`let` *decls*	(local declaration)
	|	*infixexp*	(boolean guard)

Figure 1.4: Nested declarations syntax, part 2

that has no type signature and is bound by b_2 or if this property holds transitively. A minimal set of mutually dependent bindings is called a *declaration group*. A declaration group is called *unrestricted* if and only if every variable in the group is bound by a function binding, or a pattern binding of a form *pattern* = *expression* with an explicit type signature. Otherwise, the declaration group is called *restricted*. A type variable a is considered *monomorphic* in the type t if and only if it is free in the type.

The additional monomorphism restriction besides Hindley-Milner type inference states that

- the constrained type variables of a restricted declaration group may not be generalized, and
- the monomorphic type variables that remain when the types of the entire module are inferred are *ambiguous*.

Any ambiguous types are resolved by the *defaulting* mechanism. A set of default types t_1 to t_n in the scope of the module is introduced by the language construct

`default(`t_1`, ... , `t_n`)`

and the first type that is an instance of all the classes of ambiguous variable is used.

1.5 Modules

A module is a collection of top level declarations – e. g. values, data types, classes, and instances. An environment of a module is created by a set of *imports* that

```
module    →  module modid [exports] where body
          |  body
body      →  { impdecls ; topdecls }
          |  { impdecls }
          |  { topdecls }

impdecls  →  impdecl_1 ; ... ; impdecl_n                (n ≥ 1)
topdecls  →  topdecl_1 ; ... ; topdecl_n                (n ≥ 1)

impdecl   →  import [qualified] modid [as modid] impspec

exports   →  export_1 ; ... ; export_n                  (n ≥ 1)
```

Figure 1.5: Module structure

bring top level declarations of other module into the scope. A module specifies which of the top level declarations in its scope to *export*.

A Haskell program is a set of modules. One of the modules must be called `Main` and must export the value `main`, which is the value of the program.

The syntax of a module structure is described by the Figure 1.5.

A module begins with the header `module`, which specifies the module name and a list of top level declarations to be exported. The header of the module may be omitted and the header "`module Main(main) where`" is assumed instead.

The export list *exports* may contain top level declarations or module names of imported modules. If the export list is omitted only the top level declarations in the module itself are exported. The proper description of the export list is given in The Report.

The *import* declaration brings exports of another module into the scope. Imports may be qualified and aliased. Qualified import brings only qualified names of entities into the scope, i. e., the names prefixed with the name of imported module. Aliased imports allow to import another module with local alias and qualified names are altered accordingly.

Instance declarations do not occur explicitly in export lists. All the instances in the scope of the module are always exported.

1.6 Predefined Types and Classes

In this section we describe Haskell types and classes we consider important for understanding source code examples in this thesis. Not all the types and classes are described in detail. Such description is given in The Report or in class documentation, e. g. `Traversable` [26].

1.6.1 Data Types

The Report presents standard data types as Haskell definitions although some of these definitions are not syntactically valid but convey only the meaning of the type. We do not consider important for the purpose of this thesis to distinguish built-in types from declared types.

Booleans

Boolean type is an enumeration of truth values:

```
data Boolean = False | True
```

Lists

List are data types with special syntax "`[]`".

```
data [a] = [] | a : [a]
```

Besides the special syntax lists behave as ordinary data types.

Characters and Strings

Character type `Char` is an enumeration of characters. Type `String` is a list of characters. Lexical syntax of string literals described in 1.2.3 is optional.

Unit

The unit data type is a denoted `()`. Members of the data type are nullary constructor `()` and $\perp$.

Function

Functions are data types with special syntax ($\rightarrow$) that cannot be constructed directly. Functions are constructed from already existing functions via lambda abstraction and application.

1.6.2 Classes

The Report gives a definitions of standard classes and provides a Figure 1.6 with hierarchy and instances of these classes.

Eq

The `Eq` class provides equality ($\equiv$) and inequality ($\neq$) methods.

Ord

The `Ord` class is a class of linearly ordered types. It provides the `compare` function and the $<$, $\leq$, $>$, $\geq$ operators.

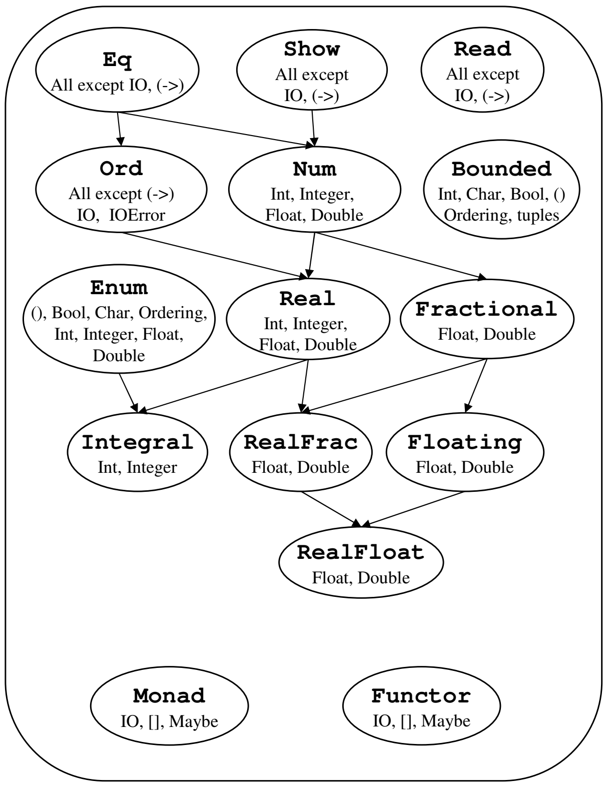

Figure 1.6: Standard Haskell classes [22]

Read and Show

The `Read` and `Show` classes are used for data types that can be converted from and to strings.

Functor

The `Functor` class is a class of data types that can be mapped over. The class provides method `fmap`. Class instances should satisfy the functor laws:

```
fmap id = id
fmap (f ∘ g) = fmap f ∘ fmap g
```

Applicative functor

The `Applicative` functor class is a subclass of the `Functor` which allows construction of functor value and application of encapsulated values of adequate types through methods `pure` and `(<$>)` respectively:

Monad

The `Monad` class is used with the do notation. The class defines methods:

$\gg$ $\ggg$= `return` `fail`

The class instances should satisfy the monadic laws:

pragma	→	`{-#` *prgdecl* `#-}`	
prgdecl	→	`INLINE` *qvar*	
	|	`NOINLINE` *qvar*	
	|	`SPECIALIZE` *spec*$_1$, … , *spec*$_k$	$(k \geq 1)$
	|	`LANGUAGE` *lang*$_1$, … , *lang*$_k$	$(k \geq 1)$
spec	→	*vars* `::` *type*	
lang	→	`Haskell2010`, …	

Figure 1.7: Pragma syntax

```
return a ≫= k = k a
m ≫= return = m
m ≫= (λx → k x ≫= h) = (m ≫= k) ≫= h
```

Standard numeric classes

The Report defines a set of numeric classes with mnemonic names. The classes are `Num`, `Real`, `Fractional`, `Integral`, `RealFrac`, `Floating`, and `RealFloat`. We do not list methods of the classes.

Foldable

The `Foldable` class is a class by Ross Paterson [30] of data structures that can folded into single value.

Traversable

The `Traversable` is a class by Conor McBride and Ross Paterson [26] of data structures, which can be traversed from left to right, performing an action on each member. It is the subclass of classes `Functor` and `Foldable`.

1.7 Compiler Pragmas

Compilers may support compiler *pragmas*, which are used to give additional information to the compiler. Lexically the pragma appears as a comment. The syntax is shown in the Figure 1.7.

The Report specifies following pragmas:

INLINE and **NOINLINE** pragmas are used to control compiler inlining.

SPECIALIZE instructs compiler to use specialized versions in order to avoid inefficiencies when dispatching overloaded functions.

$$\frac{\begin{array}{c} CE, \{u : \alpha\}, h \overset{context}{\vdash} cx : \theta \\ IE'_{sup} = vs \,\tilde{:}\, \theta \\ IE_{sup} = \forall \alpha . \Gamma \alpha \,\tilde{\Rightarrow}\, IE'_{sup} \\ \langle CE, TE \cup \{u : \alpha\}, DE \rangle \overset{sigs}{\vdash} sigs : VE_{sigs} \\ i \in [1, n] : GE, IE \oplus \{v_d : \Gamma \alpha\}, VE \overset{method}{\vdash} bind_i \rightsquigarrow \mathtt{fbind}_i : VE_i \\ VE_1 \oplus \ldots \oplus VE_n \subseteq VE_{sigs} \\ \alpha = u^{\kappa} \\ \Gamma = B^{\kappa} \\ CE' = \{B : \langle \Gamma, h, v_{def}, \alpha, IE'_{sup} \rangle\} \\ VE' = \forall \alpha . \Gamma \alpha \widetilde{\Rightarrow}_c VE_{sigs} \\ GE = \langle CE, TE, DE \rangle \\ J_{\mathtt{dict}}, \mathtt{vs}, \mathtt{v_{def}} \text{ fresh} \end{array}}{\begin{array}{ll} GE, IE, VE \overset{ctDecl}{\vdash} & \mathtt{class}\, cx \Rightarrow B\, u\, \mathtt{where}\, sigs; bind_1; \ldots; bind_n \\ & \rightsquigarrow \left\{ \begin{array}{l} \mathtt{data}\, \hat{\Gamma} = \mathtt{J_{dict}} \{ \widehat{IE'_{sup}}, \widehat{VE_{sigs}}, \}; \\ v_{def} : (\forall \alpha . \hat{\Gamma} \alpha \to \hat{\Gamma} \alpha) \\ \;= \Lambda \alpha . \lambda \mathtt{v_d} : (\hat{\Gamma} \alpha) . \mathtt{J_{dict}} \alpha \{ \mathtt{fbind}_1, \ldots \mathtt{fbind}_n \} \end{array} \right\} \\ & : \langle CE', \{\}, \{\}, IE_{sup}, VE' \rangle \end{array}}$$

Figure 1.8: Semantics of class declarations

LANGUAGE is a file header pragma that allows a particular language extension. It is up to implementation which extensions are supported, only the *Haskell2010* extension is required by The Report.

1.8 Static Semantics

In this section we give a description of static semantics of classes and interfaces as provided by Faxén in [11]. We do not provide the semantics of the rest of the language.

The static semantics of Haskell is provided as a source-to-source translation. The original language, which is syntactically described in this chapter, is referred to as the *source* language, and the language, in which the translation results, is referred to as the *target* language. The target language is a simplified variant of the source. For the purpose of this thesis we assume it to be the source language without the support of class and instance declarations.

The ad-hoc overloading is implemented by *explicit dictionary passing.* An overloaded method with type $\forall \bar{\alpha} . \theta \Rightarrow \tau$ will be translated to a function with type $\forall \bar{\alpha} . \hat{\theta} \to \tau$. The $\hat{\theta}$ is a type of a dictionary tuple for the context θ.

A class declaration of a class Γ is translated to an algebraic data type declaration for a dictionary $\hat{\Gamma}$. An instance declaration is translated into a binding for a dictionary function. The function constructs a dictionary for the given instance.

The are several environments used for carrying the information about, e. g., types, classes and instances. An environment is a set of pairs of the form *name* : *information.* There are following operations on environments:

- $dom(E_1) = \{name | name : information \in E_1\}$ is the set of names

$$\frac{i \in [1,n] : GE \overset{sig}{\vdash} sig_i : VE_i}{GE \overset{sigs}{\vdash} sig_1, \ldots, sig_n : VE_1 \oplus, \ldots, \oplus VE_n}$$

$$\frac{\begin{array}{c} KE = kindsOf(TE, CE) \\ \{u_1 : \kappa_1, \ldots, u_k : \kappa_k\} = min\{KE' | KE \oplus KE' \overset{kctx}{\vdash} cx \wedge KE \oplus KE' \overset{ktype}{\vdash} t : *\} \\ CE, TE \oplus \{u_1 : u_1^{\kappa_1}, \ldots, u_k : u_k^{\kappa_k}\}, \_ \overset{context}{\vdash} cx : \theta \\ TE \oplus \{u_1 : u_1^{\kappa_1} \ldots, u_k : u_k^{\kappa_k}\}, \_ \overset{type}{\vdash} t : \tau \\ fv(cx) \subseteq fv(t) \\ \{u | u \in dom(TE)\} \subseteq fv(t) \setminus fv(fx) \end{array}}{\langle CE, TE, DE \rangle \overset{sig}{\vdash} v :: cx \Rightarrow t : \{v : \langle v, \forall u_1^{\kappa_1} \ldots u_k^{\kappa_k} . \theta \Rightarrow \tau \rangle\}}$$

Figure 1.9: Semantics of type signatures

- $E_1 \oplus E_2$ is $E_1 \cup E_2$ assuming that $dom(E_1) \cap dom(E_2) = \emptyset$
- $kindsOf(E_1, E_2)$ is an environment that has the same kind information as the union of E_1 and E_2

There are several environments referred to:

- CE is the *class environment* that contains information about type classes. The items of the environment have the general form

  $$C : \langle \Gamma, h, \mathtt{x}_{\mathtt{def}}, IE_{sup} \rangle$$

  where Γ is the name of the class, h is an integer used to express acyclicity of the class hierarchy, $\mathtt{x}_{\mathtt{def}}$ is name of a default dictionary for the class, α is the class variable, and IE_{sup} is an instance environment described later in this list.

- TE is the *type environment* carrying the information about the type constructors and type variables.

- DE is the *data constructor* environment that describes data constructors and named fields.

- IE is the *instance environment.* It contains information on dictionary variables bound to dictionaries for instances of particular classes. There is a special notation connected with this environment – $\forall \bar{\alpha}.\theta \tilde{\Rightarrow} IE$ is a shorthand for $\{\mathtt{x} : \forall \bar{\alpha}.\theta \Rightarrow \Gamma\, \tau \in IE\}$

- VE is the *variable environment.* This environment consist of information about in-scope variables, regardless the source of the information, the information may come from algebraic data type declarations, class declarations and ordinary bindings.

There are two more operations connected to dictionaries and environments. The operator $\hat{:}$ constructs dictionary patterns from a tuple of dictionary variables

$$\frac{i \in [1,n] : CE, TE, h \overset{class}{\vdash} class_i : \Gamma_i\, \tau_i}{CE, TE, h \overset{context}{\vdash} (class_1, \ldots class_n) : \Gamma_1\, \tau_1, \ldots, \Gamma_n\, \tau_n}$$

$$\frac{\begin{array}{c} C : \langle \Gamma, h', x, \alpha, IE_{sup} \rangle \in CE \\ h' < h \\ TE, \_ \overset{type}{\vdash} u\, t_1 \ldots t_k : \tau \end{array}}{CE, TE, h \overset{class}{\vdash} C\, (u\, t_1 \ldots t_k) : \Gamma\, \tau}$$

Figure 1.10: Semantics of class and instance contexts

$(v_1, \ldots, v_n)$ (also referred to as vs) and the context θ. The other operator $\tilde{:}$ makes an instance environment from vs and θ. The set $fv(cx)$ for some context cs is a set of free variables of that context.

The source to source translation is described by a set of inference rules that

- check the program is well-formed,
- specify a translation into target language, and
- derive some information about the program in the form of environments.

The judgments have the general form

$$\textit{environment} \vdash \textit{source} \rightsquigarrow \textit{target} : \textit{derived information}$$

Class declarations

A class declaration is translated to an algebraic data type and a function. The data type is a type of dictionaries for the class and the function constructs a dictionary that contains the default methods of the class. The inference derives an information about superclasses and types of the default methods. The appropriate inference rule is described by Figure 1.8.

The class inference rule depends on inference rules *context* for superclass context, *sigs* for type signatures, and *method* for method bindings.

Type signatures

The Figure 1.9 gives judgment forms for type signatures. Type signature inference depends on kind inference *kctx* and type inference *type* rules. We do not provide these rules as we do not consider them vital for understanding the semantics of classes and instances. The rules may by looked up in [11].

Contexts

The Figure 1.10 shows inference rules for validating class and instance contexts. The integer h ensures the acyclicity of class hierarchy.

$$\frac{i \in [1,n] : GE, IE, VE \overset{instDecl}{\vdash} instDecl_i \rightsquigarrow \mathtt{binds}_i : IE_i}{GE, IE, VE \overset{instDecls}{\vdash} \left\{ \begin{array}{l} instDecl_1; \\ \ldots; \\ instDecl_n; \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \mathtt{binds}_1; \\ \ldots; \\ \mathtt{binds}_n; \end{array} \right\} : IE_1 \oplus \ldots \oplus IE_n}$$

$$\frac{\begin{array}{c} T : \chi \in TE \\ i \in [1,k] : \alpha_i = u_i^{\kappa_i} \\ C : \langle \Gamma, h, x_{def}, \alpha, IE_{sup} \rangle \in CE \\ CE, \{u_1 : \alpha_1\} \oplus \ldots \oplus \{u_k : \alpha_k\}, \_ \overset{context}{\vdash} cx : \theta \\ i \in [1,m] : GE, IE \oplus vs\tilde{:}\theta, VE \overset{method}{\vdash} bind_i \rightsquigarrow \mathtt{fbind}_i : VE_i \\ VE_{ops}[\chi\, \alpha_1 \ldots \alpha_k / \alpha] = VE_1 \oplus \ldots \oplus VE_m \\ (\forall \alpha . \Gamma\, \alpha \Rightarrow_c VE_{ops}) \subseteq VE \\ (x_1, \ldots, x_n) \tilde{:} \theta_{sup} = IE_{sup} \\ IE \oplus \mathtt{vs} \tilde{:} \theta \overset{dict}{\vdash} (e_1, \ldots, e_n) : \theta_{sup}[\chi\, \alpha_1 \ldots \alpha_k / \alpha] \\ GE = \langle CE, TE, DE \rangle \\ IE_{inst} = \{\mathtt{v}_{dict} : \forall \alpha_1 \ldots \alpha_k . \theta \Rightarrow \Gamma(\chi\, \alpha_1 \ldots \alpha_k) \\ \mathtt{vs}, \mathtt{v}_{dict} \text{ fresh} \end{array}}{\begin{array}{ll} GE, IE, VE \overset{instDecl}{\vdash} & \mathtt{instance}\, cx \Rightarrow C\, (T\, u_1 \ldots u_k) \mathtt{where}\, bind_1; \ldots; bind_m \\ & \rightsquigarrow \left\{ \begin{array}{l} \mathtt{v}_{dict}; \forall \alpha_1, \ldots, \alpha_k . \hat{\theta} \to \hat{\Gamma}\, (\chi\, \alpha_1 \ldots \alpha_k) \\ \quad = \Lambda \alpha_1, \ldots, \alpha_k . \lambda \mathtt{vs}\hat{:}\theta . \\ \quad \mathtt{let\ rec}\, v_d : \hat{\Gamma}\, (\chi\, \alpha_1 \ldots \alpha_k) \\ \qquad = (x_{def}\, (\chi\, \alpha_1 \ldots \alpha_k) v_d) \{ \\ \qquad\quad x_1 = e_1; \ldots; x_n = e_n; \\ \qquad\quad \mathtt{fbind}_1; \ldots \mathtt{fbind}_n \\ \quad \} \,\mathtt{in}\, v_d \end{array} \right\} \\ & : IE_{inst} \end{array}}$$

Figure 1.11: Semantics of instance declarations

$$\frac{\begin{array}{c} GE, IE \oplus \mathtt{vs}\tilde{:}\theta, VE \overset{bind}{\vdash} bind \leadsto \mathtt{bind} : \{x : \langle \_, \tau \rangle\} \\ \{\alpha_1, \ldots, \alpha_k\} \cap (fv(IE) \cup fv(VE)) = \varnothing \\ \mathtt{vs}\ \text{fresh} \end{array}}{\begin{array}{l} GE, IE, VE \overset{method}{\vdash} bind \leadsto \left\{ \begin{array}{ll} x = & \Lambda\, \alpha_1 \ldots \alpha_k . \lambda \mathtt{vs}\, \hat{:}\theta . \\ & \mathtt{let}\, \mathtt{bind}' \\ & \mathtt{in}\, unQual(x) \end{array} \right\} \\ \qquad : \{x : \langle x, \forall \alpha_1 \ldots \alpha_k . \theta \Rightarrow \tau \rangle\} \end{array}}$$

Figure 1.12: Semantics of method bindings

Instance declarations

An instance declaration is translated into a binding for a dictionary function, which constructs a dictionary for the declared instance. An instance environment with yielded information is returned. The rules for instance inference are described in the Figure 1.11. The instance inference depends on dictionary construction by rule *dict*.

The dictionary function produced by this translation also construct dictionaries for the respective instance of the immediate superclasses. These are listed in IE_{sup}. The dictionary function takes a tuple of dictionaries for a particular instance of the instance context and returns an appropriate dictionary.

Method bindings

Method bindings described by Figure 1.12 are used both in class declarations and instance declarations. Method bindings depend on general *bind* inference rule that is common to all bindings within the language. We do not list the *bind* rule explicitly and it may be found in [11].

Dictionary construction

Dictionary construction judgments consist of four different inference rules, which are listed in the Figure 1.13. Each rule refers to en expression `e` that evaluates to the required dictionary tuple, to the Γ_i that is the class of the dictionary, and to the types τ_i of the required instances.

The `DICT TUPLE` rule builds a dictionary tuple, the `DICT VAR` rule encloses a variable `v`, the `DICT INST` rule composes an instance dictionary from partial dictionaries, and the `DICT SUPER` rule extracts a dictionary for a superclass.

$$\text{DICT TUPLE}\;\frac{i \in [1,n] : IE \overset{dict}{\vdash} e_i : \Gamma_i\,\tau_i}{IE \overset{dict}{\vdash} (e_1,\ldots,e_n) : (\Gamma_1\,\tau_1,\ldots,\Gamma_n\,\tau_n)}$$

$$\text{DICT VAR}\;\frac{\mathtt{v} : \Gamma(\alpha\,\tau_1\ldots\tau_k) \in IE}{IE \overset{dict}{\vdash} \mathtt{v} : \Gamma(\alpha\,\tau_1\ldots\tau_k)}$$

$$\text{DICT INST}\;\frac{\begin{array}{c}\mathtt{x} : \forall\alpha_1\ldots\alpha_k.\theta \Rightarrow \Gamma(\chi\,\alpha_1\ldots\alpha_k) \in IE\\ IE \overset{dict}{\vdash} \mathtt{e} : \theta[\tau_1/\alpha_1,\ldots,\tau_k/\alpha_k]\end{array}}{IE \overset{dict}{\vdash} \mathtt{x}\,\tau_1\ldots\tau_k\,\mathtt{e} : \Gamma(\chi\,\tau_1\ldots\tau_k)}$$

$$\text{DICT SUPER}\;\frac{\mathtt{x} : \forall\alpha.\Gamma'\,\alpha \Rightarrow \Gamma\,\alpha \in IE \qquad IE \overset{dict}{\vdash} \mathtt{e} : \Gamma'\,\tau}{IE \overset{dict}{\vdash} \mathtt{x}\,\tau\,\mathtt{e} : \Gamma\,\tau}$$

Figure 1.13: Semantics of dictionary construction

Chapter 2

Maintainability Problem

In this chapter we describe a long term maintainability problem of the current type classes design. We address changes in the class system necessary to remove this problem and demonstrate the changes on examples of concrete type classes.

We also bring an overview of current possible solutions and workarounds to this problem. With the general description of the maintainability problem and particular examples of the solutions in mind, we discuss design goals of a language extension that would address the issue.

2.1 Altering Type Class Hierarchy

Type classes are one of the core features of the language, which is heavily used [39]. They also enable programmers to experiment with new approaches to data access and abstraction, e. g. the popular Lens library by Edward Kmett [21]. However, any change in a type class hierarchy requires rewriting the appropriate instance implementations. Therefore any change to the hierarchy breaks backward compatibility and thus poses significant problem to maintainability and hinders development of Haskell libraries.

It is often desirable to change class hierarchy for various reasons. In some situations the proper class relation is not understood at first and a superclass of some class is missing as is the case with `Monad` class 2.2.1. The appropriate change is being incorporate into standard library. However the process is not straightforward and brings some obstacles as we discuss bellow.

In other situations new concepts emerge and a class hierarchy needs to be refactored. John Wiegly proposed [38] to add a `Semigrupoid` class as a superclass of `Monoid` class. In the subsequent discussion Edward Kmett pointed out that this change would break existing code due to missing instances of `Semigrupoid` where the instance of `Monoid` already exists.

Other example is the `Traversable` class. It's documentation [26] specifies the behavior of instances of superclasses `Functor` and `Foldable`. Although the behavior is documented, the instances still has to be written manually and may result in erroneous code in case the violates the equivalence of, e. g., `fmap` to `fmapDefault`. It is not obvious whether this can be solved solely by refactoring the hierarchy or whether further changes are required. We address this issue in the Section 5.3.3.

As a last but not least example of standing problem regarding existing classes we give the Standard Numeric Classes. It has been argued e. g. by [36] that the standard set of numeric classes is limited in extensibility and is flawed e. g. in respect to the semantics of operations and in superfluous superclasses. However, the change in the hierarchy of these classes would break existing code.

In this section we analyze possible changes to a hierarchy of classes and state goals for a solution that deals with the problem.

In general we have identified several situations that, as we believe, may occur in the process of altering the hierarchy. Assume an arbitrary non-empty class hierarchy. Then the situations may be:

- Add a new class to the hierarchy. The class may be either standalone or connected with the rest of the hierarchy as a subclass or a superclass.
- Remove an existing class from the hierarchy. Yet again the class may be either standalone or connected to the hierarchy.
- Add a new superclass dependency or remove existing one.
- Refactor existing classes by either merging more existing classes into one or by dividing an existing class into more new classes. The new classes may be in various relations as subclasses and superclasses.
- Move a method from a subclass to a superclass or the other way around.
- Add a new class method.

In our opinion these situation cover all plausible changes to a class hierarchy. It is possible to manipulate with both classes and methods in the classes. All the situations composed of six separate actions $\mathfrak{A}$:

a_1 Add a class without methods and superclasses

a_2 Remove a class without methods and superclasses

a_3 Make an existing class a superclass of another existing class

a_4 Remove a superclass constraint from an existing class

a_5 Add a new method to some class

a_6 Remove an existing method from some class

Clearly it is possible to compose above stated situation from the set of actions $\mathfrak{A}$. Assume the first situation. By using action a_1 for the class, action a_5 repeatedly for any method of the class and action a_3 for any superclass or any class, to which it is a superclass, we get the desired situation. We do not provide compositions for the rest of the situations as we deem the process straightforward. This observation allows us to state a following theorem:

Theorem 1. *(Altering a Haskell class hierarchy)*

It is possible to compose any change in a Haskell class hierarchy only by actions from the set $\mathfrak{A}$

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

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

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

2.2 Currently Possible Solutions

In this section we describe various approaches that emerged in order to solve above stated problem. We discuss fitness of particular solutions as they tend to require some additional language extensions or adjustment to the library infrastructure.

2.2.1 Functor – Applicative – Monad

Haskell does heavily use the `Monad` class 1.6.2, which can be demonstrated on tight incorporation into the language through the `do` notation. Although it is known that every `Monad` is, in principle, a subclass of `Applicative` [12], it is not true so as current in the standard Prelude.

Changing the `Monad` class into subclass of `Applicative` would break any client code that uses the class and does not instantiate `Applicative`.

The GHC compiler takes three step process [12, 2]. In the first step the compiler was adjusted in the way that it deprecates code that does instantiate `Monad`, but does not provide an instance of `Applicative`.

In following steps the deprecated code will be fixed in libraries maintained by Haskell community and finally the superclass dependency will be added to the `Monad`.

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

2.2.2 Subclass to Superclass Instance

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

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

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

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

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

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

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

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

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

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

However, the GHC documentation [13] describes some problems that may arise with the two extensions, e. g., compiler may not be able to resolve the instance.

2.2.3 The Strathclyde Haskell Enhancement

Connor McBride has implemented as a part of his The Strathclyde Haskell Enhancement[25] (SHE) a limited version of Default Superclass Instances proposal by Jón Fairbairn [10].

The implementation is carried through preprocessor and limitations include, e. g., resolving the default instances only within the scope of one module. We consider it rather a proof of concept. However, this proof of concept together with subsequent improvements to the Fairbairn's idea, which we discuss in the Chapter 3, shows it is a viable approach.

2.3 Design Goals of Language Extension

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

The first approach (2.2.1) is not straightforward and requires manual adaptation of existing code. The second approach (2.2.2) requires some additional language extension that may cause compiler to reject the instances.

We consider McBride's SHE the most viable. In our opinion the main drawback – the limitation to one module – is caused by the fact the enhancement is implemented as a preprocessor. We believe a language extension will perform better and will also mitigate any issues of previous two approaches.

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

- The extension should allow changes to a class hierarchy where the changes are plausible[1].
- The extension should not break any existing code, i. e., the changes should be backward compatible.
- The extension should not introduce any problems to the program compilation, e. g. undecidability of instances.
- The extension should not suffer from arbitrary limitations, e. g., limitation to just one module.

[1]E. g., it is not plausible change to remove a class from a hierarchy when the class is still in use by any source code.

Chapter 3

Previous Proposals

In this section we compare different language extension proposals for type class extensions which appeared in Haskell community and attempted to solve the given problem. These were often presented only in form of a mailing list post, Haskell wikipedia entry, or similar and rendered syntax only by examples and semantics only by informal description.

Since some features are shared among different proposals we do not discuss them thoroughly with each proposal but give only summary overview of the proposal. We provide this discussion in next chapter when presenting our proposal. This chapter is solely for the purpose of giving an overview of existing proposals and presenting underlying ideas.

There has been discussion on *haskell-prime* mailing list dating back to August 2006 on availability of relationship between Functor and Monad classes in Haskell, resulting in an idea of giving default instance declarations in class declarations by Jón Fairbairn[10]. Since then several proposals which we discuss below appeared.

3.1 Default Superclass Instances by GHC

Glasgow Haskell Compiler presents proposal [8] of a language extension based on original Fairbairn's idea combined with John Meacham's Class Alias proposal [27] and Class system extension proposal [4].

The proposal states two design goals:

- the possibility to refactor class without breaking any client code
- write implementations of subclasses that imply their superclass implementations

Both goals are similar to those we presented in the Section 2.1. The proposal gives informal syntax and semantics of language extension as follows:

```
class Functor f ⇒ Applicative f where
    return :: x → f x
    (<*>) :: f (s → t) → f s → f t

    (≫) :: f s → f t → f t
    fs ≫ ft = return (flip const) <*> fs <*> ft
```

```
        instance Functor f where
            fmap = (<*>) ∘ return
```

The proposal defines **intrinsic superclass** as a transitive superclass of a class under consideration and enables to nest default instance declaration of intrinsic superclass in a class, thus the class may use its methods in definition of the instance. The proposal requires each default superclass instance to be the instance of a different class.

It further defines **intrinsic instance declaration** as an instance generated for an intrinsic superclass from class definition.

The proposal also discusses an opt-out mechanism to prevent generating of a default instance. It is done with a syntactic construct **hiding** as follows:

```
instance Sub x where
     ...
     hiding instance Super
```

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

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

Second variant is considered to be a pragmatic choice following the second design goal.

The proposal introduces language extension flag which allows defining default superclass instance. Instance of such class would not require this flag, which is in accordance with the first design goal.

The proposal shortly mentions possible feature interactions with the deriving mechanism, generic default methods, associated types, and default type synonyms.

3.1.1 Our Conclusion

In our opinion this proposal presents a nice and readable syntax of default instances extension. It also emphasizes backward compatibility of a client code which we deem desirable. The opt-out mechanisms are chosen in accordance with the design goals. However, the proposal fails to address the issue of multiple candidate intrinsic instances where a class has multiple child classes with default instances of the same method in more of them. We further elaborate on this issue in the Section 4.1.1.

We also prefer language pragma in accordance with Haskell report [22] rather than a compiler flag, though usage should be the same as with the flag, i. e., the pragma is required with a class containing a default instance but not with the instance itself.

The proposal also does not mention any use cases except those given indirectly in mechanism descriptions.

3.2 Class Alias Proposal by John Meacham

John Meacham presents a language extension proposal [27] which, as he suggests, mitigates the issues that hold back evolution of the standard prelude and provides general class abstraction capabilities. The stated goals are:

- allow modifications to a class hierarchy while retaining full backwards compatibility,
- provide both fine class granularity and practical usability with simple and advanced interfaces,
- not to interfere with separate compilation, and
- to be describable by source to source translation.

John Meacham illustrates the need for class system extension on the standard prelude and the *Num* class in particular. Given following classes:

```
class Foo a where
        foo :: a → Bool
        foo x = False

class Bar b where
        bar :: Int → b → [b]
```

he proposes a new language construct of the form:

```
class alias FooBar a = (Foo a, Bar a) where
        foo = ...
```

which declares *FooBar* as a class alias of classes *Foo* and *Bar*. The alias can occur anywhere a class constraint can. The instance of a class alias is defined in the same way as an ordinary instance:

```
instance FooBar Int where
        foo x = x > 0
        bar n x = replicate n x
```

This code defines two separate instances, one for *Foo*, the other for *Bar* and distribute class methods appropriately. With this extension in mind, *Num* class from standard prelude can be refactored, and backward compatibility is preserved by providing suitable class alias. Note that the following example slightly differs from the *Num* class as defined in Haskell2010 language report. However, we believe it does not harm its illustrative qualities:

```
class Additive a where
        (+) :: a → a → a
        zero :: a

class Additive a ⇒ AdditiveNegation where
        (-)         :: a → a → a
        negate      :: a → a
        x - y   = x + negate y
```

```
class Multiplicative a where
        (*) :: a → a → a
        one :: a

class FromInteger a where
        fromInteger      :: Integer → a


class alias Num a = (Additive a, AdditiveNegation a,
               Multiplicative a, FromInteger a)  where
        one = fromInteger 1
        zero = fromInteger 0
        negate x = zero - x
```

Authors of the proposal tender the instance of a class alias to be interchangeable with the declaration of each instance that alias constitutes of independently. However, instantiation of both alias and any of the constituent classes is illegal due to ordinary Haskell rules for overlapping instances.

3.2.1 Our Conclusion

We believe that this proposal reasonably describes requirements for changes in the class system. Nevertheless, we do not agree with class aliases as a fit approach. Though it retains backward compatibility, it is neither obvious how colliding instances should be resolved nor do the authors elaborate on that. Assume following aliases and class:

```
class alias Bar a = (Foo a, ...) where
        foo = ...

class alias Baz a = (Foo a, ...) where
        foo = ...


class (Bar a, Baz a) ⇒ Qux a where
        ...
```

In this case it is not obvious which *foo* should be used when called with the type of the class *Qux*. We also disfavor the impossibility of instantiating of both *Foo* and *Bar*. Proposal also mentions possible interactions with other type class extensions. It states that no such interaction should be an issue, although it does not bring any supporting arguments.

3.3 Superclass Defaults

The rather superficial proposal Superclass defaults [35] reflects on Meacham's class alias proposal [27]. It decomposes the original proposal into two separate issues:

- Class aliases as a single name for multiple classes

- Class method defaults in aliases

It mentions the issues of a new class like *Monad_* when providing a backward compatible default implementation of *fmap* in *Monad*, though it does not provide any details as we do in the Section 4.1.3.

The proposal consists of tho separate parts in reaction to two different issues identified in Meacham's proposal. First part deals with Superclass defaults and states as follows.

- Class declaration may contain default definitions for methods of the class or its indirect superclasses.

- An instance declaration may specify multiple classes

  ```
  instance (Class a, Class' a, ...) where
                  ...
  ```

  provided that all classes in the list are unique, type variable for all of them is same, and their superclass relation gives an acyclic graph.

- If no other implementation for a method is given the default is used. If both *Sub* and *Super* gives default implementation for a method and *Super* is superclass of *Sub* then implementation given in *Sub* is used. If more than one such an implementation exists compilation results in an error.

The other part presents class aliases extension in the same way John Meacham does. For the sake of clarity we present brief overview once again. Following syntax is introduced for the purpose of defining a class alias:

```
class alias Ctx ⇒ Alias a = (Class1 a, Class2 a, ..) where
        ...
```

The body may contain default implementations of methods from the alias definition and their superclasses. In any context, *Alias a* is treated the same as a class list. In an instance head,*Alias a* is treated the same as *(_Alias a, Class1 a, Class2 a, ...)*, where *_Alias* is a fresh name that cannot be directly referred to, treated as a subclass of classes in alias definition and containing default methods from the class alias body.

3.3.1 Our Conclusion

We agree with the clear distinction between class aliases and superclass defaults as presented. We deem class aliasing substantial for backward compatibility when altering class hierarchy for reasons we present in the Section 4.1.3. The example of the artificial *Monad_* alias is illustrative. We do not favor multiple class instantiation, since we neither consider it helpful for programmer in the terms of the clarity of the code, nor is it necessary for backward compatibility. We also consider particular mechanism for selecting which class is to be used where multiple definitions of the same method in the hierarchy exits to be reasonable, though it is not described in detail. We dislike the method definition both in a class and an alias. We consider the definition in the alias superfluous and believe that wherever there is a need for such definition, a proper class should be introduced.

3.4 Class System Extension Proposal

This class system extension proposal [4] adduces an idea—similar to the previously mentioned proposals—that it is often possible to provide a default implementation to a class method not only by using other class methods but also by using methods of class ancestors. Given example classes

```
class Functor m where
        fmap :: (a → b) → m a → m b

class Functor m ⇒ Applicative m where
        return :: a → m a

        apply :: m (a → b) → m a → m b

        (≫) :: m a → m b → m b
        ma ≫ mb = (fmap (const id) ma) ‘apply‘ mb


class Applicative m ⇒ Monad m where
        (≫=) :: m a → (a → m b) → m b
```

the proposal summarises reasons to do so as follows.

- It is necessary to provide an instance definition only for the class *Monad*, the other two can be derived.
- Many existing programs don't provide *Applicative* instance when instantiating *Monad* and making *Monad* subclass of *Applicative* would break existing code.
- A method implementation depends on a particular subclass and it may not by possible to provide such an implementation in class where the method is introduced. An example of *fmap* is given:

```
class Applicative m ⇒ Monad m where

        fmap f ma = ma ≫= (λa → return (f a))
```

Based on these observation extension proposes following changes in Haskell class system:

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

- Modules export only specific instance declarations.

Proposal argues that separate compilation is possible due to the fact that compiler generated instances which are supplemented to explicit instances are visible only in the module being compiled. Proposal also observes that the resolution of an overloaded method depends on the visible instances in the module where method is called. Therefore, overloading needs to be resolved before merging the modules together, in particular inlined method overloading needs to be resolved before the method is inlined.

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

Proposal also introduces a feature of explicit import and export of instances, in order to restrain colliding instance declarations among different packages. Proposed syntax is

```
module M (
        -- exported instances
          instance Monad T
        , instance Functor (F a) hiding (Functor (F Int), Functor (F Char))
        , F(..)
) where

import Foo (instance Monad a hiding Monad Maybe)

data T a
data F a b
```

where the context is omitted because it is not used in instance selection. The import directive instructs the compiler to use all instances of *Monad* exported by *Foo* except instance for *Monad Maybe*, regardless of this instance being exported.

The proposal also endorses the need for class aliases in order to make complex types manageable, but refers to the Class Aliases proposal [27] and the Superclass defaults proposal [35] we described in the Section 3.2 and Section 3.3 respectively.

The proposal further mentions Extensible superclases, based on a paper by Sulzmann and Wang [33], and serve rather a purpose of maintainability of the code than the issue under our consideration and are beyond the scope of our thesis, we do not provide further description.

3.4.1 Conclusion

We favor the part of the proposal related to the issue of backward compatible superclass defaults. We consider the provided propositions of instance visibility and selection sound, though we do not consider clarification of possibility of separate compilation sufficient. We do believe the Class Aliases to be necessary for backward compatible changes in the class hierarchy and, as this proposal is just referring to the proposals [27] and [35], our conclusion in sections 3.2 and 3.3 holds.

We do not consider the explicit import and export of instances to be necessary for superclass defaults, though it might be possibly beneficial. We consider Extensible type classes and Quantified contexts beyond the scope of our thesis.

Chapter 4

Language Extension Proposal

We have given an overview of existing proposals we regard relevant to the problem under consideration and based on merits of these proposals we derive a new Haskell language extension. We believe this approach results in a reasonable combination of desirable properties mitigating unwanted issues.

In the first section of this chapter we give a summary of different features presented across more proposals. Then we present our own version of a language extension in form of two new language constructs. I the End of this chapter we describe several use cases the language constructs we propose.

4.1 Language Extension Features

Proposals presented in the Chapter 3 particularly overlap and share some features. We consider following features significant:

Default instances as described in 3.1

Default method implementation as described in 3.3 and 3.4

Class aliases as described in 3.2, 3.3, and 3.4

4.1.1 Default Instances

Default instances proposal is presented in a single proposal, unlike the other two aforelisted features. Thus our conclusion in the Section 3.1 covers the aspects of this feature. We consider it to be a suggestive way of describing alternative implementation of superclass' method in subclass due to its resemblance with ordinary instance definition. However, we want to elaborate on the issue of multiple candidate intrinsic instances.

Assume following **Library** and associated **Client** code:

```
module Library where


class Foo a where
        method :: a → a

class Foo a ⇒ Bar a where
```

```
        ...
        instance Foo a where
                method = ...


class Bar a ⇒ Baz a where
        ...
        instance Foo a where
                method = ...


class Bar a ⇒ Qux a where
        ...
        instance Foo a where
                method = ...



module Client where

import Library

data MyData = ...

instance Bar MyData where
        ...

instance Baz MyData where
        ...

instance Qux MyData where
        ...

myFunction :: MyData → MyData
myFunction = method
```

This example shows the class hierarchy and client code that results in multiple candidate default (intrinsic) instances. When using `method` on `MyData` that is the instance of above classes it is not obvious which default instance to use. Possible candidates are instances generated in `Bar`, `Baz`, and `Qux`. We consider a pragmatic choice to select such instance that is:

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

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

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

4.1.2 Default Method Implementation

Presented proposals endorse the idea of providing default implementation of the class method of class or any indirect superclasses in class alias in case of the Superclass Defaults Proposal 3.3 and providing default implementation of any ancestor method directly in class in case of Class System Extension Proposal 3.4. This approach may derive suitable method definition from subclass instance as shown by example in the Section 3.4.

This approach further requires to provide implicitly generated instances in a case we add a superclass to a class. Assume following module **Module** which defines classes `Foo`, `Bar`, and `Baz`.

```
module Module where

class Foo a where
        foo :: a

class Foo a ⇒ Bar a where
        bar :: a

class Baz a where
        baz :: a
```

And a client code which imports `Module` and uses the class `Baz`:

```
module ClientCode where

import Module

data MyData = ...

instance Baz MyData where
        baz = ...
```

If we want to add the class context of `Bar`to the class `Baz`in such manner this change does not break the existing client code we need to provide instance of `Bar MyData` and `Foo MyData` transitively. The solution provided in 3.4 is to always generate automatically full set of instance declarations for all superclasses. This solution does not however deal with another problem. Assume that we the alter `Module` in the following manner:

```
module Module where

class Foo a where
        foo :: a
```

```
class Foo a ⇒ Bar a where
        bar :: a

        -- default method for Foo.foo
        foo = ...

class Bar a ⇒ Baz a where
        baz :: a

        -- default method for Foo.foo
        foo = ...
```

When using this version of the module providing that default instances are generated it is not obvious which version of `foo` should be used. Neither of the proposals deal with the issue. Possible options may be:

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

The first option is not favorable as it may be the case that the implementation in the class `Baz` has more specific information therefore its implementation has better performance. The second option requires reasonable policy for selecting one implementation among all candidates even in case of more elaborate class hierarchies than the hierarchy in the example.

Another issue we want to point out is that providing default method implementation for superclass method contradicts some design recommendations, e. g. we do not consider such an implementation trivial [7].

4.1.3 Class Aliases

Class aliases are presented in three different proposals in the Chapter 3. Aliases are proposed both as a mean of providing default method implementation and single name for multiple classes. We consider the later vital for retaining backward compatibility. This problem is hinted in proposal 3.3. Assume following classes `Applicative` and `Monad`:

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

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

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

```
data Foo = ...

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

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

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

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

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

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

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

This works fine considering original code which uses `Monad` and does not instantiate `Applicative`. However we now have two different names for `Monad` and it is not obvious which one should user who is aware of their existence use.

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

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

class Bind a where3
        (>>=)   :: m a → (a → m b) → m b
        (>>)    :: m a → m b → m b
        m >> k  = m >>= λ _ → k
```

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

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

The other significant feature of class alias is that it enables programmer to provide single instance that gives declarations of methods of multiple classes. This in turn allows instantiation without the necessity of renaming of the methods. Assume two related classes `Tweedledum` and `Tweedledee` [3] and their class alias `Rumdumdee`:

```
class Tweedledum a where
        dum = ...

class Tweedledee a where
        dee = ...

class alias Rumdumdee a = (Tweedledum a, Tweedledee a)

...

instance Rumdumdee MyData where
        dum = ...
        dee = ...
```

The two instances for alias are generated accordingly. Without the alias functionality the `Rumdumdee` must by a class with superclasses `Tweedledum` and `Tweedledee`. The instances for these two may be defaulted, but methods `dum` and `dee` must be renamed. We do not see this as favorable.

4.2 Our Proposal

Based on the conclusions in the Chapter 3 and the observations in previous section we derive our own proposal which aims to address issues described in the Section 2.1, i. e.:

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

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

When adding a superclass into a context of subclass we distinguish several situations. Assume two classes, a class `Foo` and `Bar`, that were not (indirect) subclass of each other and we want to add the `Bar` into the context of `Foo` such that `Foo` is a subclass of the class `Bar`. Assume that there is already a client code where Foo is brought into the scope and then subsequently used, i. e. some instance is provided. Then if there is also an instance of Bar in the client code everything works fine.

On the other hand when the instance of Bar is missing compilation results in an error message[1] similar to:

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

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

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

The first approach requires an active change in the class definition whereas later results in an error when method without definition is used.

The last situation which can occur when adding the class Bar into the context of the class Foo is that only the class Bar is used in a client code. In this situation the client code works without any problems.

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

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

Removing of an existing class that is neither superclass not subclass of any other class is possible as long as the class is not used. In the case the class is rendered obsolete it should be deprecated by a pragma mechanism [13]. But in cases where the class is to be deleted as a result of the other action discussed, e. g. the class was split into two, it is necessary to provide it both for backward compatibility and for use in new code. It is possible to provide a class of the same name with the classes in which it was split as superclasses and default implementation original methods–which are now in superclasses. The other way is to provide a class alias of the same name as the original class aliasing the classes in which it was split. We consider the second approach superior as it result in shorter, more readable code.

[1]The example error message is produced with *The Glorious Glasgow Haskell Compilation System, version 7.6.3*

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

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

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

Our proposal consist of two independent language modifications. We devise a mechanism for providing default instances of superclasses and mechanism of class aliasing. We describe these modifications in the two separate chapters.

4.3 Superclass Default Instances

We propose to add a new syntax construct into the class definition as described in the Section 1.4.3. Programmer may provide a *default instance* in the class method for any of it's superclasses, e. g.:

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

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

4.3.1 Syntax of the Extension

We described the Haskell type class system in the Section 1.4.3. With respect to the proposed change we adjust formal description of class declarations as shown in the Figure 4.1.

A class declaration contains nested default instance declaration of the general form:

topdecl	→	`class` [*scontext* `=>`] *tycls* *tyvar* [`where` *cdecls*]	
scontext	→	*simpleclass*	
	|	`(` *simpleclass₁* `,` … `,` *simpleclassₙ* `)`	$(n \geq 0)$
simpleclass	→	*qtycls* *tyvar*	
cdecls	→	`{` *cdecl₁* `;` … `;` *cdeclₙ* `}`	$(n \geq 0)$
cdecl	→	*gendecl*	
	|	(*funlhs* | *var*) *rhs*	
	|	`default instance` *qtycls* *dinst* [`where` *didecls*]	
dinst	→	*gtycon*	
	|	`(` *gtycon* *tyvar₁* … *tyvarₖ* `)`	($k \geq 0$, *tyvars* distinct)
	|	`(` *tyvar₁* `,` … `,` *tyvarₖ* `)`	($k \geq 2$, *tyvars* distinct)
	|	`[` *tyvar* `]`	
	|	`(` *tyvar₁* `->` *tyvar₂* `)`	(*tyvar₁* and *tyvar₂* distinct)
didecls	→	`{` *idecl₁* `;` … `;` *ideclₙ* `}`	$(n \geq 0)$
didecl	→	(*funlhs* | *var*) *rhs*	
	|		(*empty*)

Figure 4.1: Class declarations

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

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

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

$$di_i = D\,a$$

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

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

We do not change the declaration of ordinary instance. However there can be both ordinary instance and either one or multiple *default instances* in the same scope. There cannot be more ordinary instances due to overlapping instances restriction. We consider the graph of all classes in scope. This graph is required to be acyclic by [22] and has natural ordering $\prec'$ generated by class dependencies.

$$C \prec' C$$

and

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

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

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

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

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

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

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

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

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

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

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

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

4.3.2 Semantics of the Extension

In this section we provide static semantics of our extension. Default instances require a change on the `ctDecl` inference rule described in the Figure 4.2. This change requires new inference rule `dinstDecls`.

The new inference rules `dinstDecls` and `dinstDecl` transform default instances into the same representation as original rules transforms ordinary instances. These rule are also modeled in a similar way to `instDecls` and `instDecl` respectively.

$$\frac{\begin{array}{c} CE, \{u : \alpha\}, h \overset{context}{\vdash} cx : \theta \\ IE'_{sup} = vs \tilde{:} \theta \\ IE_{sup} = \forall\alpha.\Gamma\alpha \mathrel{\tilde{\Rrightarrow}} IE'_{sup} \\ \langle CE, TE \cup \{u : \alpha\}, DE\rangle \overset{sigs}{\vdash} sigs : VE_{sigs} \\ i \in [1, n] : GE, IE \oplus \{v_d : \Gamma\alpha\}, VE \overset{method}{\vdash} bind_i \leadsto \mathtt{fbind}_i : VE_i \\ VE_1 \oplus \ldots \oplus VE_n \subseteq VE_{sigs} \\ \alpha = u^{\kappa} \\ \Gamma = B^{\kappa} \\ cs' = B\, u \\ CE, IE \oplus IE_{sup}, CE, VE, cx' \overset{dinstDecls}{\vdash} didecls \leadsto \mathtt{dibinds} : IE_{di} \\ CE' = \{B : \langle\Gamma, h, v_{def}, \alpha, IE'_{sup}\rangle\} \\ VE' = \forall\alpha.\Gamma\alpha \widetilde{\Rrightarrow}_c VE_{sigs} \\ GE = \langle CE, TE, DE\rangle \\ J_{\mathtt{dict}}, \mathtt{vs}, \mathtt{v}_{\mathtt{def}} \text{ fresh} \end{array}}{\begin{array}{ll} GE, IE, VE \overset{ctDecl}{\vdash} & \mathtt{class}\, cx \Rightarrow B\, u\, \mathtt{where} \\ & sigs; \\ & bind_1; \ldots; bind_n; \\ & didecls; \\ & \leadsto \left\{ \begin{array}{l} \mathtt{data}\, \hat{\Gamma} = \mathtt{J}_{\mathtt{dict}}\{\widehat{IE'_{sup}}, \widehat{VE_{sigs}}, \}; \\ v_{def} : (\forall\alpha.\hat{\Gamma}\alpha \to \hat{\Gamma}\alpha) \\ \ = \Lambda\alpha.\lambda\mathtt{v}_{\mathtt{d}} : (\hat{\Gamma}\alpha).\mathtt{J}_{\mathtt{dict}}\alpha\{\mathtt{fbind}_1, \ldots \mathtt{fbind}_n\}; \\ \mathtt{dibinds} \end{array} \right\} \\ & : \langle CE', \{\}, \{\}, IE_{sup} \oplus IE_{di}, VE'\rangle \end{array}}$$

Figure 4.2: New semantics of class declarations

$$\frac{i \in [1,n] : GE, IE, VE, cx \overset{dinstDecl}{\vdash} dinstDecl_i \rightsquigarrow \mathtt{binds}_i : IE_i}{GE, IE, VE, cx \overset{dinstDecls}{\vdash} \left\{ \begin{array}{l} dinstDecl_1; \\ \ldots; \\ dinstDecl_n; \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{l} \mathtt{binds}_1; \\ \ldots; \\ \mathtt{binds}_n; \end{array} \right\} : IE_1 \oplus \ldots \oplus IE_n}$$

$$\frac{\begin{array}{c} T : \chi \in TE \\ i \in [1,n] : \alpha_i = u_i^{\kappa_i} \\ C : \langle \Gamma, h, x_{def}, \alpha, IE_{sup} \rangle \in CE \\ CE, \{u_1 : \alpha_1\} \oplus \ldots \oplus \{u_n : \alpha_n\}, \_ \overset{context}{\vdash} cx : \theta \\ i \in [1,m] : GE, IE \oplus vs \tilde{:} \theta, VE \overset{method}{\vdash} bind_i \rightsquigarrow \mathtt{fbind}_i : VE_i \\ VE_{ops}[\chi\, \alpha_1 \ldots \alpha_n / \alpha] = VE_1 \oplus \ldots \oplus VE_m \\ (\forall \alpha . \Gamma\, \alpha \Rightarrow_c VE_{ops}) \subseteq VE(x_1, \ldots, x_n) \tilde{:} \theta sup = IE_{sup} \\ IE \oplus \mathtt{vs} \tilde{:} \theta \overset{dict}{\vdash} (e_1, \ldots, e_n) : \theta_{sup}[\chi\, \alpha_1 \ldots \alpha_n / \alpha] \\ GE = \langle CE, TE, DE \rangle \\ IE_{inst} = \{ v_{dict} : \forall \alpha_1 \ldots \alpha_n . \theta \Rightarrow \Gamma(\chi\, \alpha_1 \ldots \alpha_n / \alpha) \\ \mathtt{vs}, \mathtt{v}_{dict} \text{ fresh} \end{array}}{\begin{array}{ll} GE, IE, VE, cx \overset{dinstDecl}{\vdash} & \mathtt{default\ instance}\, C\, (T\, u_1 \ldots u_k)\mathtt{where} \\ & \quad bind_1; \ldots; bind_m \\ & \rightsquigarrow \left\{ \begin{array}{l} \mathtt{v}_{dict}; \forall \alpha_1, \ldots, \alpha_k . \hat{\theta} \to \hat{\Gamma}\, (\chi\, \alpha_1 \ldots \alpha_k) \\ \quad = \Lambda \alpha_1, \ldots, \alpha_k . \lambda \mathtt{vs} \hat{:} \theta . \\ \quad \mathtt{let\ rec}\, v_d : \hat{\Gamma}\, (\chi\, \alpha_1 \ldots \alpha_k) \\ \quad\quad = (x_d ef\, (\chi\, \alpha_1 \ldots \alpha_k) v_d) \{ \\ \quad\quad\quad x_1 = e_1; \ldots; x_n = e_n; \\ \quad\quad\quad \mathtt{fbind}_1; \ldots \mathtt{fbind}_n \\ \quad \} \, \mathtt{in}\, v_d \end{array} \right\} \\ & : IE_{inst} \end{array}}$$

Figure 4.3: Semantics of default instance declarations

topdecl	$\rightarrow$	`classalias` [*scontext* `=>`] *tycls tyvar*	
scontext	$\rightarrow$	*simpleclass*	
	$\mid$	($simpleclass_1$, … , $simpleclass_n$)	$(n \geq 0)$
simpleclass	$\rightarrow$	*qtycls tyvar*	

Figure 4.4: Class alias declarations

4.4 Class Aliases

We propose to add a new syntax construct into the class definition as described in the Section 1.4.3. Programmer may provide a *class alias* for several different classes, e. g.:

```
classalias  (Read a, Show a) ⇒ Textual a
```

The class alias may be instantiated by usual instance declaration. Compiler generates separate instances for `Read` and `Show` classes and distributes the class method accordingly.

4.4.1 Syntax

Class aliases are new top level declaration. We present changes in formal syntax in the Figure 4.4.

A class alias declaration has a general form:

```
classalias cx ⇒ D u
```

This introduces new *class alias* of the classes in the context *cx*. The class in context are required to form an acyclic directed graph and must be all different. The aliased classes may not contain methods with the same name.

The declaration of *class alias* may contain binding only for the class methods of aliased classes. If no binding is given for some method default method in the class declaration is used. It there is no such method the method of the instance is bound to *undefined*.

4.4.2 Relation to The Superclass Default Instances

Note that the class aliases are not necessary for use of superclass default instances. However, class aliases make some changes in class hierarchy easier and more direct. Assume following classes in a library code

```
class C a where
        ...

class D a where
        method :: a
```

and instance in a client code:

```
instance D MyData where
        method = ...
```

Without class aliases `method` cannot be moved from `D` to `C` contemporary with making `C` the superclass of `D`. The client instance expects `D` to have the `method` and we have not changed this anyhow. For sure it is possible to introduce some proxy method `method'` and provide following version of the library:

```
class C a where
        ...
        method' :: a

class C a ⇒ D a where
        method :: a
        default instance C a where
                method = method'
```

Nevertheless, with class aliases following solution is feasible:

```
class C a where
        ...
        method :: a

classalias (C a) ⇒ D a
```

We consider second approach superior as it does not introduce new, duplicate methods.

Chapter 5

Relation to the Language Platform

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

5.1 Relation to the Existing Language Extensions

The Haskell language has evolved beyond the specification provided by The Report. The GHC, upon which we build our implementation (see Chapter 6), contains variety of additional language extensions that can be looked up in [13].

We consider important to assess possible clashes with other extensions even though proper testing should be provided. Nevertheless, such testing is extremely time consuming due to the great number of the extensions[1]. In sight of this fact we do not list all the extension but only the extension we evaluated as possible clashes during the implementation. E. g. our implementation does not seem to be involved with any of the syntactic extension anyhow and thus we do not consider these here.

5.1.1 Multi-parameter Type Classes

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

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

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

5.1.2 Default Method Signatures

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

5.1.3 Functional Dependencies

The `FunctionalDependencies` extension allows programmer to constrain parameters of type classes. This requires the `MultiParamTypeClasses` extension to be active thus same approach as in the case of later extension applies. This extension is not supported.

5.1.4 Flexible Instances and Undecidable Instances

These two extensions relax constraints on instance context. A context of a superclass default instance is dependent on a class where is this instance declared and is described in the Section 4.3. This context is restricted enough in the terms of Haskell 2010 and further relaxation on constraints does not involve our extension.

5.2 Comparison of Current Solutions to Default Superclass Instances

In this section we provide alternatives to the solutions presented in the Section 2.2. We discuss merits of our extension compared to the original solution.

5.2.1 Backward Incompatible Change in Hierarchy

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

```
{-# LANGUAGE SuperclassDefaultInstances #-}

...

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

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

        default instance Applicative m where
```

```
                pure a = return a
                (<*>) :: f (a → b) → f a → f b
                f <*> a = a ≫= (λx → fmap (λg → g x) f)
```

This change in the library is backward compatible thus it can be deployed immediately. The advantage is there is no transitional period with deprecation we discussed in the Section 2.2.1.

5.2.2 Subclass to Superclass Instance

Our solution allows programmer to provide the same functionality as the solution in the Section 2.2.2. Unlike that solution our approach does not require problematic `UndecidableInstances` extension.

5.2.3 SHE

Our solution is conceptually similar to SHE and thus are both solutions comparable. Unlike SHE our extension has full information available to the compiler and does not suffer from limitations, e. g., to one source file.

5.3 Applications of Proposed Extension

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

5.3.1 Bind and Pointed

Edward Kmett has pointed out [20] that although current classes `Functor`, `Applicative`, and `Monad` may be refactored into more convenient structure from categorical point of view it is not to the benefit of programmer without some kind of superclass default instance mechanism. The new structure is shown in Figure 5.3.1. Our mechanism allows such refactoring. It is in principle corresponding to the solution given in the Section 5.2.2:

```
{-# LANGUAGE SuperclassDefaultInstances #-}

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

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

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

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

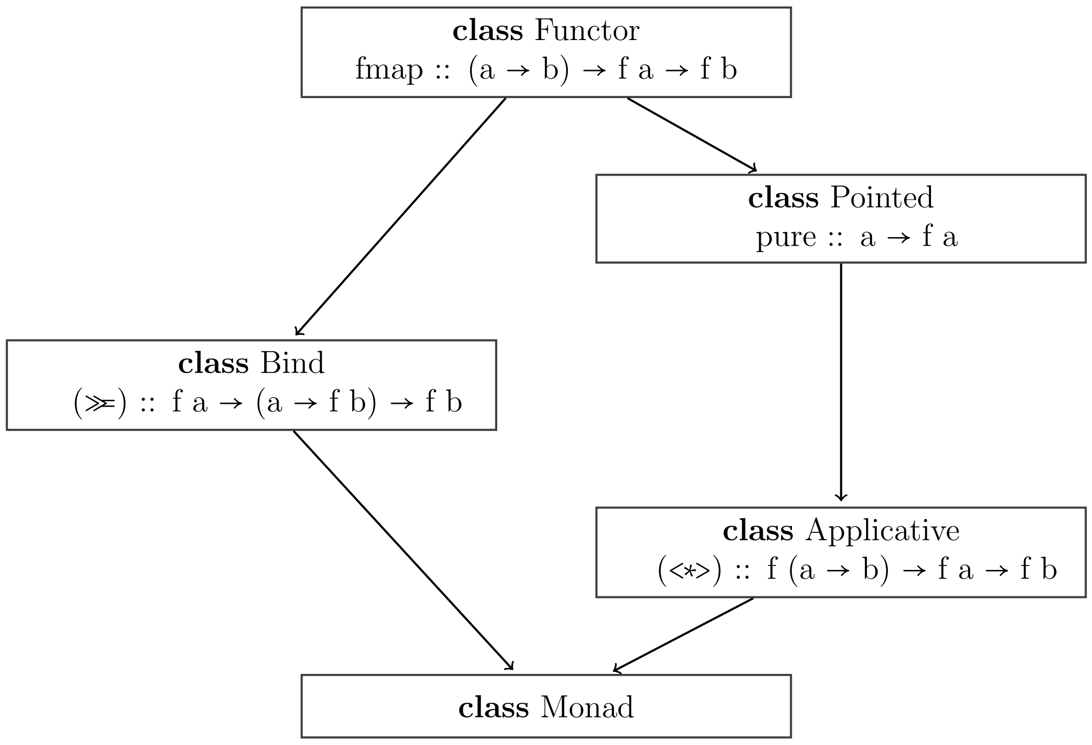
class Functor
fmap :: (a → b) → f a → f b
class Pointed
pure :: a → f a
class Bind
(≫=) :: f a → (a → f b) → f b
class Applicative
(<*>) :: f (a → b) → f a → f b
class Monad

Figure 5.1: Refactored class structure

```
classalias (Bind m, Applicative m) ⇒  Monad m
```

The use of class alias here allows user of the library to provide instance of `Monad` with methods ≫=, <*> and `pure`, which are distributed accordingly. Note that this set of methods is not a set of method usually assumed with `Monad` and we use it here for illustrative purposes.

5.3.2 Standard Numeric Classes

There has been discussion on the design of standard numeric classes. According to [36] there are several problems—one of them that standard classes are not finely-grained enough. With our extension it is possible to refactor current structure in a more apt one and maintain backward compatibility.

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

```
{-# LANGUAGE SuperclassDefaultInstances #-}
...

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

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

class (Eq a, Show a) ⇒ Num a where
```

```
        (+), (*) :: r → r → r

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

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

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

5.3.3 Traversable

The documentation of `Traversable` package [26] currently states properties that instance of the class is expected to satisfy with respect to the classes `Foldable` and `Functor`. However, it is up to the programmer to ensure this. It is possible with the Superclass Default Instances extension to provide instances satisfying these rules automatically and thus avoid possible inconsistencies:

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

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

newtype Const a = Const { getConst :: a }

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

class (Functor t, Foldable t) ⇒ Traversable t where
        ...
        default instance Functor t where
                fmap f = getId ∘ traverse (Id ∘ f)
        default instance Foldable t where
                foldMap = getConst ∘ traverse (Const ∘ f)
```

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

Chapter 6

Implementation

In this section we briefly describe an implementation details of the language proposal we described in the Section 4. We have selected the GHC as a compiler into which we incorporate our extension. According to the [18] it is currently the only compiler that supports Haskell 2010 specification. We also consider the GHC [13] documentation on compiler internals and compiler development superior to other compilers (e. g. Utrecht Haskell Compiler [9]).

We provide only an implementation of the Superclass Default Instances extension. It is possible to avoid the use of class aliases with proxy methods as we described in section 4.4.2. Thus we consider the implementation of first of the extension sufficient to asses contributions of our work before eventual incorporation of these extension into GHC.

6.1 Compiler Architecture

In this section we refer chiefly to description of the compiler by the main authors, Marlow and Peyton-Jones, in [23]. They state modularity and the openness to the research and compiler extension to be one of the project goals. The modularity allows us to described the architecture and consecutively the changes to the compiler in several detached steps.

In general the GHC project involves more than just the compiler itself. Precisely it contains

- the compiler,
- the basic libraries that the compiler depends upon, and
- The Runtime System (RTS) that handles running the compiled code.

We do not need to take care of the architecture of neither libraries nor the RTS. The libraries contain general data structures, e. g. `Data.Map`, and although we make use of them, we do not need to make any changes. The compiler processes source program into *Haskell core*, a variant of system F called FC [34].

Any higher syntactic construct are translated into the core at first and the code generation follows after this process. Thus we do not need to take the RTS into account when describing our implementation as no adjustments to it take place.

The compiler is further divided into three parts:

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

Only the Hsc is of our concern as we do not change behavior of multiple file compilation nor manipulate with any external programs.

6.1.1 The Haskell Compiler

Compiling a Haskell source file proceeds sequentially in several phases. The structure of the phases is illustrated in the Figure 6.1.1

Parser

In this phase the Haskell source file is converted into abstract syntax. The lexical analyser and parser are involved. The abstract syntax data type is the `HsSyn t` for some type of identifier `t`. The parses produces the original string names `RdrName` from source code. Hence, the type of abstract syntax is `HsSyn RdrName`.

This part of compiler uses the Alex library [1] for generating lexical analyser and the Happy library [15] as a parser generator.

Renamer

This compilation phase all identifiers into the fully qualified names. The identifier types are resolved from `RdrName` to references to particular entity `Name`. Therefore, the resulting abstract syntax has type `HsSyn Name`.

Type checker

The type checker verifies that the Haskell program is type-correct. The type checker resolves the types of identifiers resulting in conversion of `Name` type in abstract syntax to `Id` type.

Desugaring and simplification

Desugaring translates all higher language constructs into basic core language. Then the core code is simplified through optimization, e. g., dead code elimination and case expression reduction.

After these phases the target code generation follows. Particular behavior depends on compiler settings–either native machine code, LLVM code, or C code may be produced. In this process GHC also generates interface files in order to support separate compilation.

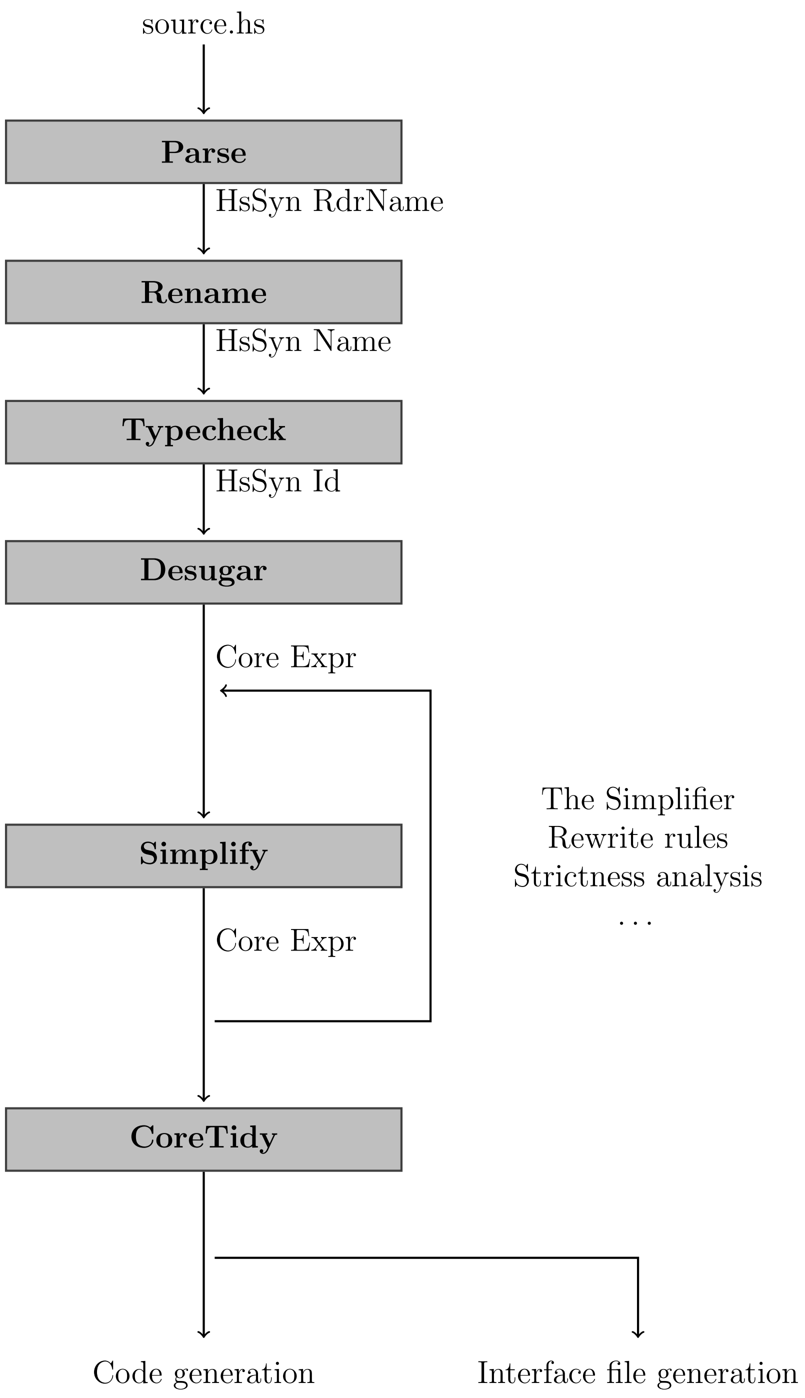

Figure 6.1: The compilation phases

6.2 Changes to the Compiler

In this section we briefly document changes to the compiler and design of our implementation. In the description of the changed we follow sequential architecture of the compiler and discuss each part of the compiler respective to the compilation phase separately.

The compiler allows extensive testing output through command line options. We found this output most instructive when testing the changes and list samples cases in the Appendix C.

6.2.1 Extension Flag

Before proceeding with the implementation have we added the *Extension Flag* and registered the flag with a command line option. This allows programmer to enable the extension from source code and from the command line respectively. The appropriate data types are located in the file `main/DynFlags.hs`.

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

...

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

This allows us to recognise whether the extension is enable from compilation context.

6.2.2 Parser

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

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

and introduced default instance rule:

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

Note that the `hintSuperclassDefaultInstances` is a new helper that checks the extension flag and causes appropriate compilation error if the flag is not present.

The effect of this change on the parser is demonstrated in the example C.1. For technical reasons we require user to explicitly state the context of default instance.

6.2.3 Renamer

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

6.2.4 Type Checker

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

```
data ClsDefInstDecl name = ClsDefInstDecl {
  cdid_poly_ty :: LHsType name    -- Context ⇒ Class Instance-type
-- Using a polytype means that the renamer conveniently
-- figures out the quantified type variables for us.
, cdid_binds :: LHsBinds name
, cdid_sigs  :: [LSig name] -- User-supplied pragmatic info
, cdid_tyfam_insts :: [LTyFamInstDecl name] -- type family instances
, cdid_datafam_insts :: [LDataFamInstDecl name] -- data family instances
, cdid_overlap_mode :: Maybe OverlapMode
}
deriving (Data, Typeable)
```

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

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

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

6.2.5 Desugaring

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

6.3 Concluding notes

We have given a brief overview of our implementation. This implementation is to be consider only a proof-of-concept. In our description we elaborate only on changes in the main data structures of the compiler. We do not give any description of subsequent changes in the code. E. g. many changes in the code for debugging output are trivial and we do not consider such description to be necessary.

Further examples of testing source code are included with our implementation.

Conclusion

We have described a maintainability problem with Haskell type classes that occurs in practice. We have summarized previous attempts to solve this problem and analysed different features of this approaches.

Based on this analysis we have derived a proposal for two language extensions—the Superclass Defaults Extension and Class Aliases extensions. The first proposal allows programmer to declare default instance of a superclass within a class declaration. The other extension allows to provide an instance over a set of classes. We have demonstrated on several examples how the superclass default instances solve the problem under our consideration. The Class Aliases are only a supplementary extension to Superclass Defaults but works nicely with it and allows cleaner expression of some class hierarchies.

We have discussed relations of our extensions to existing extensions and provided sample solutions to discussed problem. We have also implemented the first of the extensions as a proof-of-concept and described the main design choices of this implementation.

The given implementation of our proposal show that our proposal is solid and can be implemented in practice. Listed sample solutions only cover a small set of problems whether the real application of our work could be much wider. Introduction of these language extensions into mainstream compiler would allow on one hand to correct some existing problems in class dependencies and on the other hand the authors of libraries would have more freedom in the design of such libraries knowing that any design choice can be corrected in future.

Further work

The future work related to this thesis embodies mostly in two directions. There is a great deal of work ahead consisting of a proper implementation of the extensions. Such implementation must adhere to GHC coding standards in order to be acceptable into the compiler. Such implementation must also contain extensive test cases for use with GHC testing platform. Following such implementation and its acceptance into compiler there are changes to be made in the standard library with the use of our extensions.

Beside the practical work there is also the other direction. The more theoretical direction of the work consists of possible opt-out mechanism for default instances. We do not discuss these in out work although they are considered in the previous proposals. There are also some other language proposals that are loosely related to the default instances, e. g. quantified contexts [32]. Identification of these proposals and their relation to superclass defaults seems to us as a solid base for further work on the language extension that the language and its users could benefit from.

Bibliography

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

[2] *Applicative/Monad proposal related warnings.* Online. July 2014. URL: `https://ghc.haskell.org/trac/ghc/ticket/8004`.

[3] Lewis Carroll. *Through the Looking-Glass, and What Alice Found There.* London: Macmillan, 1871.

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

[5] FP Complete CORP. *Integrated Analysis Platform.* Online. July 2014. URL: `https://www.fpcomplete.com/business/iap/`.

[6] Luis Damas. "Type Assignment in Programming Languages". PhD thesis. University of Edinburgh, 1984.

[7] *Default method implementation.* URL: `http://www.haskell.org/haskellwiki/Default_method_implementation`.

[8] *Default superclass instances.* Online. July 2014. URL: `https://ghc.haskell.org/trac/ghc/wiki/DefaultSuperclassInstances?version=30`.

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

[10] Jón Fairbairn. *All Monads are Functors.* Haskell prime mailing list. Aug. 2006.

[11] Karl-Filip Faxén. "A static semantics for Haskell". In: *Journal of Functional Programming* 12 (2002), pp. 295–357.

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

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

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

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

[16] Roger Hindley. “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146 (Dec. 1969), pp. 29–60.

[17] Paul Hudak et al. “A History of Haskell: Being Lazy with Class”. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages.* HOPL III. San Diego, California: ACM, 2007, pp. 1–55.

[18] *Implementations.* Online. Jan. 2014. URL: `http://www.haskell.org/haskellwiki/index.php?title=Implementations&oldid=57412`.

[19] Simon L. Peyton Jones and Philip Wadler. *A Static Semantics for Haskell.* Tech. rep. 1991.

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

[21] Edward Kmett. *The lens package.* Online. 2014. URL: `http://hackage.haskell.org/package/lens-4.3.2`.

[22] Simon Marlow. *Haskell 2010 Language Report.* Tech. rep. June 2010. URL: `http://www.haskell.org/onlinereport/haskell2010/`.

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

[24] Simon Marlow et al. “The Haxl Project at Facebook”. In: *Proceedings of the Code Mesh London.* 2013.

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

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

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

[28] Robin Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[29] Bryan O’Sullivan, Don Stewart, and John Goerzen. *Real World Haskell.* O’Reilly, Nov. 2008.

[30] Ross Paterson. *Data.Traversable.* Online. 2005. URL: `https://hackage.haskell.org/package/base-4.7.0.0/docs/Data-Foldable.html`.

[31] Iustin Pop. “Experience report: Haskell as a reagent: results and observations on the use of Haskell in a python project”. In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming.* New York, NY, USA, 2010, pp. 369–374.

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

[33] Martin Sulzmann and Meng Wang. “Modular generic programming with extensible superclasses”. In: *ICFP-WGP.* 2006, pp. 55–65.

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

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

[36] *The numeric-prelude package*. July 2014. URL: `http://hackage.haskell.org/package/numeric-prelude-0.4.1`.

[37] *The Other Prelude*. Online. Dec. 2010. URL: `http://www.haskell.org/haskellwiki/index.php?title=The_Other_Prelude&oldid=37992`.

[38] John Wiegley. *Proposal: Add Data.Semigroup to base, as a superclass of Monoid*. June 2013.

[39] Brent Yorgey. "The Typeclassopedia". In: *The Monad.Reader* 13 (2009), pp. 17–69.

List of Figures

1.1 Class declaration syntax 13
1.2 Instance declaration syntax 13
1.3 Nested declarations syntax, part 1 14
1.4 Nested declarations syntax, part 2 15
1.5 Module structure 16
1.6 Standard Haskell classes [22] 18
1.7 Pragma syntax 19
1.8 Semantics of class declarations 20
1.9 Semantics of type signatures 21
1.10 Semantics of class and instance contexts 22
1.11 Semantics of instance declarations 23
1.12 Semantics of method bindings 24
1.13 Semantics of dictionary construction 25

4.1 Class declarations 46
4.2 New semantics of class declarations 48
4.3 Semantics of default instance declarations 49
4.4 Class alias declarations 50

5.1 Refactored class structure 55

6.1 The compilation phases 59

Appendix A

Lexical Structure

For the sake of completeness we present the lexical structure of the Haskell programming language. The presented syntax uses the same notational conventions as described in The Report [22]:

[*pattern*	optional
{*pattern*}	zero or more repetitions
(*pattern*)	grouping
$pat_1 \mid pat_2$	choice
$pat_{(pat')}$	difference - elements generated by *pat*, but not by *pat'*
`fibonacci`	terminal syntax

Production rules are given in the BNF-like format. In the lexical syntax the "maximal munch" rule is used, i. e. the grammatical phrases extend as far to the right as possible.

program	→	{ *lexeme* | *whitespace* }
lexeme	→	*qvarid* | *qconid* | *qvarsym* | *qconsym*
	|	*literal* | *special* | *reservedop* | *reservedid*
literal	→	*integer* | *float* | *char* | *string*
special	→	`(` | `)` | `,` | `;` | `[` | `]` | `` ` `` | `{` | `}`
whitespace	→	*whitestuff* {*whitestuff*}
whitestuff	→	*whitechar* | *comment* | *ncomment*
whitechar	→	*newline* | *vertab* | *space* | *tab* | *uniWhite*
newline	→	*return linefeed* | *return* | *linefeed* | *formfeed*
return	→	a carriage return
linefeed	→	a line feed
vertab	→	a vertical tab
formfeed	→	a form feed
space	→	a space
tab	→	a horizontal tab
uniWhite	→	any Unicode character defined as whitespace
comment	→	*dashes* [$any_{\langle symbol \rangle}$ {*any*}] *newline*
dashes	→	`--` {`-`}
opencom	→	`{-`
closecom	→	`-}`

ncomment	→	*opencom ANYseq* {*ncomment ANYseq*} *closecom*	
ANYseq	→	$\{ANY\}_{\langle \{ANY\}\ (\ opencom \mid closecom\ )\ \{ANY\} \rangle}$	
ANY	→	*graphic* | *whitechar*	
any	→	*graphic* | *space* | *tab*	
graphic	→	*small* | *large* | *symbol* | *digit* | *special* | `"` | `'`	
small	→	*ascSmall* | *uniSmall* | `_`	
ascSmall	→	`a` | `b` | ... | `z`	
uniSmall	→	any Unicode lowercase letter	
large	→	*ascLarge* | *uniLarge*	
ascLarge	→	`A` | `B` | ... | `Z`	
uniLarge	→	any uppercase or titlecase Unicode letter	
symbol	→	*ascSymbol* | $uniSymbol_{\langle special \mid \_ \mid " \mid ' \rangle}$	
ascSymbol	→	`!` | `#` | `$` | `%` | `&` | `*` | `+` | `.` | `/` | `<` | `=` | `>` | `?` | `@`	
	|	`\` | `^` | `|` | `-` | `~` | `:`	
uniSymbol	→	any Unicode symbol or punctuation	
digit	→	*ascDigit* | *uniDigit*	
ascDigit	→	`0` | `1` | ... | `9`	
uniDigit	→	any Unicode decimal digit	
octit	→	`0` | `1` | ... | `7`	
hexit	→	*digit* | `A` | ... | `F` | `a` | ... | `f`	
varid	→	$(small\ \{small \mid large \mid digit \mid '\ \})_{\langle reservedid \rangle}$	
conid	→	*large* {*small* | *large* | *digit* | `'` }	
reservedid	→	`case` | `class` | `data` | `default` | `deriving` | `do` | `else`	
	|	`foreign` | `if` | `import` | `in` | `infix` | `infixl`	
	|	`infixr` | `instance` | `let` | `module` | `newtype` | `of`	
	|	`then` | `type` | `where` | `_`	
varsym	→	$(\ \mathbf{symbol}_{\langle : \rangle}\ \{\mathbf{symbol}\}\ )_{\langle reservedop \mid dashes \rangle}$	
consym	→	$(\ :\ \{\mathbf{symbol}\})_{\langle reservedop \rangle}$	
reservedop	→	`..` | `:` | `::` | `=` | `\` | `|` | `<-` | `->` | `@` | `~` | `=>`	
varid			(variables)
conid			(constructors)
tyvar	→	*varid*	(type variables)
tycon	→	*conid*	(type constructors)
tycls	→	*conid*	(type classes)
modid	→	{**conid** `.`} *conid*	(modules)
qvarid	→	[*modid* `.`] *varid*	
qconid	→	[*modid* `.`] *conid*	
qtycon	→	[*modid* `.`] *tycon*	
qtycls	→	[*modid* `.`] *tycls*	
qvarsym	→	[*modid* `.`] *varsym*	
qconsym	→	[*modid* `.`] *consym*	

decimal	→	*digit*{*digit*}
octal	→	*octit*{*octit*}
hexadecimal	→	*hexit*{*hexit*}
integer	→	*decimal*
	|	`0o` *octal* | `0O` *octal*
	|	`0x` *hexadecimal* | `0X` *hexadecimal*
float	→	*decimal* `.` *decimal* [*exponent*]
	|	*decimal* *exponent*
exponent	→	(`e` | `E`) [`+` | `-`] *decimal*
char	→	`'` ($graphic_{\langle \texttt{'} \mid \backslash \rangle}$ | *space* | $escape_{\langle \backslash \texttt{\&} \rangle}$) `'`
string	→	`"` {$graphic_{\langle \texttt{"} \mid \backslash \rangle}$ | *space* | *escape* | *gap*} `"`
escape	→	`\` (*charesc* | *ascii* | *decimal* | `o` *octal* | `x` *hexadecimal*)
charesc	→	`a` | `b` | `f` | `n` | `r` | `t` | `v` | `\` | `"` | `'` | `&`
ascii	→	`^`*cntrl* | `NUL` | `SOH` | `STX` | `ETX` | `EOT` | `ENQ` | `ACK`
	|	`BEL` | `BS` | `HT` | `LF` | `VT` | `FF` | `CR` | `SO` | `SI` | `DLE`
	|	`DC1` | `DC2` | `DC3` | `DC4` | `NAK` | `SYN` | `ETB` | `CAN`
	|	`EM` | `SUB` | `ESC` | `FS` | `GS` | `RS` | `US` | `SP` | `DEL`
cntrl	→	*ascLarge* | `@` | `[` | `\` | `]` | `^` | `_`
gap	→	`\` *whitechar* {*whitechar*} `\`

Appendix B

Syntax of the Language

module	→	`module` *modid* [*exports*] `where` *body*	
	$\mid$	*body*	
body	→	`{` *impdecls* `;` *topdecls* `}`	
	$\mid$	`{` *impdecls* `}`	
	$\mid$	`{` *topdecls* `}`	
topdecls	→	$topdecl_1$ `;` … `;` $topdecl_n$	$(n \geq 1)$
topdecl	→	`type` *simpletype* `=` *type*	
	$\mid$	`data` [*context* `=>`] *simpletype* [`=` *constrs*] [*deriving*]	
	$\mid$	`newtype` [*context* `=>`] *simpletype* `=` *newconstr* [*deriving*]	
	$\mid$	`class` [*scontext* `=>`] *tycls* *tyvar* [`where` *cdecls*]	
	$\mid$	`instance` [*scontext* `=>`] *qtycls* *inst* [`where` *idecls*]	
	$\mid$	`default` `(`$type_1$ `,` … `,` $type_n$`)`	$(n \geq 0)$
	$\mid$	`foreign` *fdecl*	
	$\mid$	*decl*	
decls	→	`{` $decl_1$ `;` … `;` $decl_n$ `}`	$(n \geq 0)$
decl	→	*gendecl*	
	$\mid$	(*funlhs* $\mid$ **pat**) *rhs*	
cdecls	→	`{` $cdecl_1$ `;` … `;` $cdecl_n$ `}`	$(n \geq 0)$
cdecl	→	*gendecl*	
	$\mid$	(*funlhs* $\mid$ *var*) *rhs*	
idecls	→	`{` $idecl_1$ `;` … `;` $idecl_n$ `}`	$(n \geq 0)$
idecl	→	(*funlhs* $\mid$ *var*) *rhs*	
	$\mid$		(empty)
gendecl	→	*vars* `::` [*context* `=>`] *type*	(type signature)
	$\mid$	*fixity* [*integer*] *ops*	(fixity declaration)
	$\mid$		(empty declaration)
ops	→	op_1 `,` … `,` op_n	$(n \geq 1)$
vars	→	var_1 `,` … `,` var_n	$(n \geq 1)$
fixity	→	`infixl` $\mid$ `infixr` $\mid$ `infix`	

Appendix C

Debugging Outputs

In this appendix we list debugging outputs of the compiler after different phases of compilation. This outputs demonstrate changes to the different parts of the GHC we described in the Section 6.2.

All the outputs use following simple source file `DSI.hs`:

```
{-# LANGUAGE SuperclassDefaultInstances #-}
module SDI where

class Super a where
        foo :: a

class Super b ⇒ Sub b where
        bar :: b
        default instance Sub b ⇒ Super b where
                foo = bar

data Unit = Unit

instance Sub Unit where
        bar = Unit
```

C.1 Parser Output

The compiler output with the *ddump-parsed* option:

```
# ./ghc-stage2 --ddump-parsed DSI.hs
[1 of 1] Compiling SDI              ( ./DSI.hs, ./DSI.o )

==================== Parser ====================
module SDI where
class Super a where
  foo :: a
class Super b => Sub b where
  bar :: b
  default instance {-# NO_OVERLAP #-} Super b where
    foo = bar
```

```
data Unit = Unit
instance Sub Unit where
  bar = Unit
```

C.2 Renamer Output

The compiler output with the *ddump-rn* option:

```
# ./ghc-stage2 --ddump-rn DSI.hs
[1 of 1] Compiling SDI              ( ./DSI.hs, ./DSI.o )

==================== Renamer ====================
class SDI.Super a where
  SDI.foo :: a
class SDI.Super b => SDI.Sub b where
  SDI.bar :: b
  default instance {-# NO_OVERLAP #-} SDI.Super b where
    SDI.foo = SDI.bar
data SDI.Unit = SDI.Unit

instance SDI.Sub SDI.Unit where
  SDI.bar = SDI.Unit
```

C.3 Type Checker Output

The compiler output with the *ddump-tc* option:

```
# ./ghc-stage2 --ddump-tc DSI.hs
[1 of 1] Compiling SDI              ( ./DSI.hs, ./DSI.o )
TYPE SIGNATURES
TYPE CONSTRUCTORS
  class Super b => Sub b where
    bar :: b
  class Super a where
    foo :: a
  data Unit = Unit
    Promotable
COERCION AXIOMS
  axiom SDI.NTCo:Super :: Super a = a
INSTANCES
  instance Sub Unit -- Defined at ./SDI.hs:16:10
  instance Sub b => Super b -- Defined at ./SDI.hs:11:26
DEFAULT INSTANCES
  instance Sub b => Super b -- Defined at ./SDI.hs:11:26
  instance Sub Unit -- Defined at ./SDI.hs:16:10
  instance Sub b => Super b -- Defined at ./SDI.hs:11:26
Dependent modules: []
Dependent packages: [base, ghc-prim, integer-gmp]
```

```
=================== Typechecker ====================
AbsBinds [] []
  {Exports: [SDI.$fSubUnit <= $dSub_aqn
              <>]
   Exported types: SDI.$fSubUnit [InlPrag=[ALWAYS] CONLIKE]
                     :: Sub Unit
                   [LclIdX[DFunId],
                    Str=DmdType,
                    Unf=DFun: \ -> SDI.D:Sub TYPE Unit $dSuper_aqj $cbar]
   Binds: $dSub_aqn = SDI.D:Sub $cbar}
AbsBinds [] []
  {Exports: [$cbar <= bar
              <>]
   Exported types: $cbar :: Unit
                   [LclId, Str=DmdType]
   Binds: AbsBinds [] []
            {Exports: [bar <= bar
                        <>]
             Exported types: bar :: Unit
                             [LclId, Str=DmdType]
             Binds: bar = SDI.Unit}}
AbsBinds [b] [$dSub_aqo]
  {Exports: [SDI.$fSuperb <= $dSuper_aqw
              <>]
   Exported types: SDI.$fSuperb [InlPrag=INLINE (sat-args=0)]
                     :: forall b. Sub b => Super b
                   [LclIdX[DFunId(nt)], Str=DmdType]
   Binds: $dSuper_aqw = SDI.D:Super ($cfoo)}
AbsBinds [b] [$dSub_aqo]
  {Exports: [$cfoo <= foo
              <>]
   Exported types: $cfoo :: forall b. Sub b => b
                   [LclId, Str=DmdType]
   Binds: AbsBinds [] []
            {Exports: [foo <= foo
                        <>]
             Exported types: foo :: b
                             [LclId, Str=DmdType]
             Binds: foo = bar}}
```