Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Bialgebraic Description of Generalized Binary Methods and Other Topics in the Semantics of Object-Oriented Languages

Candidate:
Rekha Redamalla

Supervisor:
Furio Honsell
Marina Lenisa

April 6, 2007

Author's e-mail: redamall@dimi.uniud.it

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

To my daughters,
Sathwika and Divyagna

Abstract

Object-Oriented Languages support a very flexible programming paradigm for implementing data abstraction. However, there does not exist as yet an established approach to a formal semantics for such languages. This is unsatisfactory, since a formal semantics is crucial for developing full fledged formal methods for any programming language.

The present thesis provides contribution towards a robust development of a coalgebraic semantics for Object-Oriented Languages. In particular, it addresses the critical issue of dealing with binary methods. These are methods that take more than one parameter of a class type. Although this problem has been dealt with before in the literature, we feel that it has not yet been solved in a completely satisfactory way. The appeal of coalgebraic methods is their considerably low mathematical over head and their operational nature. The solution we propose, we think fits naturally in this spirit.

More specially: we extend the H.Reichel and B.Jacobs coalgebraic account of specification and refinement of objects and classes in Object Oriented Programming to (*generalized*) *binary methods*. Generalized binary methods are methods whose type parameters are built over constants and class variables, using products, sums and powerset type constructors. In order to take care of class *constructors*, we model classes as *bialgebras*. We study and compare two solutions for modelling generalized binary methods, which use, somewhat surprisingly, only, purely covariant functors. In the first solution, which applies when we already have a class implementation, we reduce the behaviour of a generalized binary method to that of a bunch of unary methods. These are obtained by *freezing* the types of the extra class parameters to constant types. The *bisimulation behavioural equivalence* induced on objects by this model amounts to the *greatest congruence* with respect to method application. Alternatively, in the second solution, we treat binary methods as *graphs* instead of functions, thus turning contravariant occurrences in the functor into covariant ones.

We investigate a coalgebraic semantics for the class-based object oriented imperative language, Fickle, introduced and studied by Drossopoulou et al.. Fickle is a `Java`-like language, extending `Java` with object *re-classification*. First we investigate notions of *observational equivalences* over *Fickle* programs. Then, in order to study them, we define a coalgebraic model for Fickle programs. However, in order to deal with the store, we need to extend the original approach of H.Reichel and B.Jacobs, which is purely functional.

Finally, we address the problem of typing binary methods when subclasses are considered as subtypes. We propose yet another solution, inspired by a view of functions as graphs, based on a new typing system, where one can annotate in the type of an object whether a method is never called on that object.

Acknowledgments

I am deeply indebted to my supervisors, Prof. Furio Honsell and Prof. Marina Lenisa for their guidance, insight and encouragement throughout the course of my doctoral program and for their kindness. Marina Lenisa, who has been an immensesource of inspiration and constant support at each and every stage for a successful culmination of this dissertation. She has been exceptionally generous.

I would like to express my gratitude to all the members of research group of Formal Methods and Logics of Computation group for the numerous discussions and cooperation, and all the members of DIMI University of Udine for being so helpful in my Research.

I would like to thank Prof. Furio Honsell, Vice-Chancellor of the University of Udine and Dr.B.G Sidharth, Directer General of B.M Birla Science Center, Hyderabad, for the joint-research collaboration project under EU-India Economic Cross Cultural Programme (ECCP), ICT for EU-India Cross Cultural Dissemination - ALA/95/23/2003/077-54, by which I got opportunity to pursue research, and International Institute for Applicable Mathematics and Information Sciences (IIAMIS), a joint institute between B.M Birla Science Center, Hyderabad-India and University of Udine-Italy for motivation and support.

Special thanks go to external referees, Prof. Jan Rutten, CWI Amsterdam, The Netherlands and Dr. Luigi Liquori, Maitre de Recherche, INRIA Sophia Antipolis, France, for their careful reading, valuable comments and suggestions about the writing of this manuscript.

I would not have embarked on pursuing this scientific enquiry without the constant support and encouragement of my daughters, Sathwika and Divyagna, I am indebted to my husband, Sri Sathyanarayan Reddy Bavikadi, who has been an immensesource of inspiration and constant support for a successful culmination of this dissertation. I would especially like to thank to my Parents and in-laws for their encouragement.

This work is partially supported by the UE project DART: Dynamic Assembly, Reconfiguration and Type-checking - IST-2001-33477, TYPES - IST-CA-510996, the MIUR Project CoMeta: Computational Metamodels and ART: Analysis of systems of Reduction by means of Systems of Transition - PRIN 200501824.

Contents

List of Figures

List of Tables

Introduction

Coalgebraic semantics originated with Aczel-Mendler, Rutten-Turi, for CCS-like languages, [3, 4, 70, 71, 72], and it was further generalized to λ-calculus, [41], higher-order imperative languages, [51], Object-Oriented languages in a functional setting, [64, 47, 49], π-calculus, [43, 16] by various other researchers: Honsell, Jacobs, Lenisa, Reichel.

The gist of the *coalgebraic semantics paradigm* (*final semantics*) is to view the *interpretation* function from *syntax* to *semantics* as a *final* mapping in a suitable category. To this end, the semantics has to be construed as a *final coalgebra* for a suitable functor F and the syntax has to be cast in form of an F-coalgebra. This approach is driven by the operational semantics of the language, because it is the semantics which determines the structure of the functor F. This is dual to the syntax-driven approach of *algebraic semantics* (*initial, denotational semantics*), where syntax is construed as an initial F-algebra and the semantics is defined as an F-algebra. The main advantage of the coalgebraic semantics is that it induces a *behavioural equivalence* on programs, which can be characterized as a *coalgebraic bisimilarity*, i.e. as greatest coalgebraic bisimulation.

In [64, 47, 49], Reichel and Jacobs have introduced a coalgebraic model of objects and classes of Object-Oriented languages (OO-languages). The idea underpinning this approach is that coalgebras, duals to algebras, allow to focus on the behaviour of objects while abstracting from the concrete representation of the state of the objects. Algebras have "constructors" (operations packaging information into the underlying carrier set); coalgebras have "destructors" or "observers" (operations extracting information out of the carrier set), which allow to detect certain behaviours.

Classes in OO-languages are given in terms of *attributes* (*fields*) and *methods*. The values of attributes determine the states of the *class*, *i.e.* the *objects*; methods act on objects.

In the coalgebraic approach of [64, 47, 49], a class is modelled as an F-coalgebra $(A, f : A \to F(A))$ for a suitable functor F. The carrier A represents the space of *attributes*, or *fields*, and the coalgebra operation f represents the *public* methods of the class, i.e. the methods which are accessible from outside the class. Thus the objects of a class are modelled as the elements of the carrier. Their behaviour under application of public methods, viewed as functions acting on objects, is then captured by the coalgebra map f. Thus the coalgebraic model induces exactly this behavioural equivalence on objects, whereby two objects are equated if, for each public method, the application of the method to the two objects, for any list of parameters, produces equivalent results. A benefit of the coalgebraic model is a coinduction principle for establishing the behavioural equivalence.

The original Reichel-Jacobs approach has been mainly used to study the correspondence between class specifications (i.e. abstract classes together with assertions on method's behaviour) and class implementations in the field of program and data refinement, [49, 39].

But in Reichel-Jacobs approach, only a single class in *isolation* is modelled and the setting is purely *functional*. Moreover, binary methods, i.e. methods which take another instance of the hosting class as argument[1], cannot be described in such original coalgebraic approach, since they would produce a contravariant occurrence of the variable in the corresponding functor.

The present thesis provides contribution towards a robust development of a coalgebraic semantics for OO-languages. In particular, it addresses the critical issue of dealing with binary methods. Although this problem has been dealt with before in the literature, we feel that it has not yet been solved in a completely satisfactory way. The appeal of coalgebraic methods is their considerably low mathematical over head and their operational nature. The solution we propose, we think fits naturally in this spirit. More specially: in this thesis, we extend Reichel-Jacobs coalgebraic description to *generalized binary methods*, *i.e.* methods whose type parameters are built over constants and class variables, using products, sums and powerset type constructor. This is a quite

[1] By a standard abuse of terminology, *binary methods* refer to methods with $n \geq 2$ class parameters.

large collection of methods, including all the methods which are commonly used in Object-Oriented Programming, and potentially more.

Our focus of interest are equivalences on objects which are "well-behaved", *i.e.* are *congruences w.r.t.* method application. We propose two different solutions. Our first solution is based on the observation that the behaviour of a generalized binary method can be captured by a bunch of unary methods obtained by *"freezing"*, in turn, the types of the class parameters to the states of the class implementation given at the outset, *i.e.* by viewing them as constant types. Our second solution is based on a set-theoretic understanding of functions, whereby binary methods in a class specification are viewed as *graphs* instead of functions. Thus contravariant function spaces in the functor are rendered as covariant sets of relations.

We prove that the behavioural equivalence induced by the *"freezing approach"* amounts to the *greatest congruence w.r.t.* method application on the given class, at least for *finitary* binary methods, *i.e.* methods where the type constructors range over *finite* product, sum and powerset. As a by-product, we provide a (coalgebraic) coinduction principle for reasoning about such greatest congruence.

As far as the graph model is concerned, the behavioural equivalence is not a congruence, even for finitary binary methods. However, we show that a natural necessary and sufficient condition for this to hold is that the graph and freezing equivalence coincide. As a consequence, when this is the case, we obtain a spectrum of logically independent coinduction principles for reasoning on the greatest congruence.

In this thesis, we also investigate the possibility of extending the original coalgebraic approach to imperative OO-languages. To this aim, we focus on a fragment of the imperative typed class-based language *Fickle*, [29, 30], which extends `Java` with *re-classification*. Re-classification allows objects to change class membership dynamically (*e.g.* see [22, 74, 33, 75]), while retaining their identity.

In dealing with Fickle, the approach of [64, 47] needs to be refined to accommodate *imperative* features as well as general *programs*, i.e. sequences of classes possibly related by inheritance, mutual definitions, etc. Special care needs to be devoted to representing the *store*, and in defining the evolution of objects, we have to take into account all possible pointers involving them.

In this thesis, we deal directly only with *unary* methods, and we discuss the extra problematic issues which arise in the imperative setting, when we try to model also binary methods. Moreover, in this thesis, we investigate the possibility of using the coalgebraic model also for *program equivalence* and *program transformation*. This is some what dual goal *w.r.t.* the program refinement of Reichel and Jacobs.

Finally, we discuss the problem of typing binary methods, when subclasses are considered as subtypes. In concrete OO-languages, the problem is either solved by only altering *overriding* and forbidding *overloading* (such as in C++) or by implementing method calls with *multiple dispatching* (such as in `Java`), *i.e.* by choosing the method code to activate according to the types of all class parameters, and *not* only the object type. In this thesis, we propose an alternative solution based on *single dispatching* and on a new typing system, where one can annotate in the type of an object whether a method is never called on that object, this solution is suggested by the graph coalgebraic semantics.

Summing up, the main contributions of this thesis are:

1. a coalgebraic semantics of generalized binary methods, main results are published in [45];

2. a coalgebraic semantics of imperative OO-Languages, main results are published in [44];

3. a typing system for typing binary methods, in presence of a subtyping relation.

While we feel that the objective in (1) is substantially achieved, apart from the case of infinitary methods, (2) and (3) require further developments before the problems can be considered and solved in full generality.

Structure of the Dissertation

Here we outline the structure of the dissertation and the origin of each chapter. Apart from the present introduction, this thesis is divided into three main parts.

In Part I, we present Object Oriented Programming concepts and Examples of Object Oriented languages. This part consists of two chapters. General principles of OO-Programming, abstract data types and types for Object Oriented language are discussed in Chapter 1. An example of a class-based, Object Oriented imperative language, namely Fickle, is presented in Chapter 2 together with its syntax, operational semantics and typing rules. Moreover, in this chapter, we introduce and discuss various notions of observational equivalences.

In Part II, we present the theory of co(bi)algebras. This part consists of two chapters. In Chapter 3, we describe bialgebraic preliminaries, *i.e.* we introduce the basic notions of algebras, coalgebras, bialgebras, congruence, bisimulation, and final semantics. In Chapter 4, we present basic concepts of (co)algebraic specifications. We will introduce the notions of class specification and class implementation, together with various examples of bialgebraic specification.

In Part III, we present the issues related to binary methods. This part contains the main contribution of the thesis and it consists of three chapters. In Chapter 5, we present our bialgebraic model of classes and objects of OO-languages. this extends the original Reichel-Jacobs coalgebraic model to generalized binary methods. In Chapter 6, we present our co(bi)algebraic account of Fickle programs, together with natural observational equivalences and we discuss the extra issues arising in presence of binary methods. In Chapter 7, we discuss the problem of typing binary methods when subclasses are considered as subtypes. We discuss some known solutions, and we present a new solution inspired by a view of functions as special graph relations.

In the conclusions, we discuss remarks, open questions and possible directions for future work. In Appendix 7.3, we present categorical preliminaries and theorems.

I

First Part: Object-Oriented Programming

1

General Principles of Object-Oriented Programming

In this chapter we provide an introduction and discuss the main issues concerning the Object-Oriented paradigm. In particular we discuss Object-Oriented Programming, abstract data types, and types for Object-Oriented languages. We follow rather closely W. Cook [24], B. Meyer [60], M. Abadi et al. [1].

1.1 Object-Oriented Programming

In Object-Oriented Programming (OO-Programming for short), one fundamental notion is that of object. Abstractly, an object is a *datum* together with operations that can query and modify its state (i.e. actual data + processing). We can think of an object as state together with a collection of operations, often referred to as methods (or procedures, that share access to private local state - actual processing). In implementations, sets of operations are often associated with classes of objects and it is merely an illusion that each object has its own embedded operations.

Most Object-Oriented languages (OO-languages) can be class-based or object-based. OO-languages, such as *Smalltalk*, C++, `Java`, *Fickle* and *OCaml*, use classes to create objects, are called *class-based* languages. In such languages, the implementation of an object is specified by its *class* and the objects are created by instantiating their classes. There are also some OO-languages with objects but no classes, are called *object-based* languages like the experimental language *Self* [79], or the *ECMAscript* language. In *object-based* languages, such as *Self*, the effect of classes can be achieved by creating a canonical object for each class and using cloning to create new instances.

The general principles of object oriented programming are: objects; data abstraction; classes; inheritance; polymorphism and dynamic binding; and multiple inheritance. Here is a brief summary of object-oriented features [60].

- *Data abstraction* describes the ability to hide implementation details of one part of a program from another and focuses on the meaning of the operations (behaviour). In the context of objects, this often means that a program can only access an object's state indirectly via its methods. Thus we effectively reduce the size of an object's interface. Small interfaces result in more easily maintainable code. The notion of data abstraction is not exclusive to OO programming. It is key to the concept of abstract data types. For examples of data abstraction in a non-object setting, see [62].

- *Classes* can be considered as templates for creating objects. Classes define the shape of the internal state and the methods for objects, its instances, created from it. Other than giving compilers hints as to how to group code for methods, it is also a convenient way for inducing a type system for objects.

- *Inheritance* is a feature that aids reuse of software components. It is typically available in languages with classes, and allows programmers to define new classes by extending existing

classes - these can share their behaviour without having to reimplement that behaviour. The new class is called an immediate *subclass* of the existing class which is referred to as the immediate *superclass*. More generally, we use the term subclass to refer to the transitive closure of immediate subclass, and similarly for superclass. Inheritance saves effort from the programmer's point of view because the subclass's methods are copied, or *inherited,* from the superclass.

Subtyping allows instances of subclasses to be used in places where an instance of its superclass is expected. For languages whose types are induced by classes, the subtype relation is usually identified with inheritance. In such languages, subtyping with dynamic binding gives a form of polymorphism that makes inheritance a particularly useful feature.

- *Polymorphism* allows the same code to be applied to objects of different classes, so long as they support a common interface. With subtyping, this allows existing code to be easily reused.

- *Dynamic binding* is the use of the pseudo-variable `self`, which refers to the current object, in method-code to refer to sibling methods. In the presence of inheritance, dynamic binding significantly increases the expressiveness of the language. In the absence of inheritance, the sibling methods are always the same. With inheritance, this is no longer the case since a subclass can inherit some methods, and override or overload (see Section 1.4) others. Dynamic binding allows method-code to always invoke the correct sibling methods.

- *Multiple inheritance* is the ability to derive subclasses from more than one superclass. Its position as an essential feature of an OO-language is in constant debate.

When information systems are modeled as objects, they can employ the powerful inheritance capability. For instance, instead of building separate tables for employees with department and job information, the type of employee is modeled. The employee class contains the data and the processing for all employees. Each subclass (manager, secretary, etc.) contains the data and processing unique to that person's job. Changes can be made globally or individually by modifying the appropriate class.

Inheritance can be implemented using *delegation*, where a subclass can be approximated by a new object that owns an instance of the superclass, or *embedding*, where attributes and/or methods of one object are copied into another.

Object-based languages are possibly more expressive than *class-based* languages. For example, in an *object-based* language, one can create objects whose shape is determined at runtime. Conversely, in a traditional *class-based* language, objects are instances of classes which cannot be created at runtime i.e. dynamically. Class-based languages are more established in industry than their object-based counterparts. Object-based languages (like *Self*) are used particularly in the area of artificial intelligence.

1.2 Abstract Data Types

The earlier concepts of "data type" and "data structure" have gradually merged into the single concept of data abstraction. This merger has occurred as an outgrowth of research in a number of areas related to the structure and meaning of programs, including the design of programming languages, the theory of programming language semantics, the verification of programs, and the theory of data types and structures.

Abstract data types(ADT's) are the traditional way of treating data abstraction. The basic concept of abstraction is that a data type should be defined only in terms of operations that are valid on objects of its type, not in terms of how the type is implemented on a real computer. Objects of a given type are created and inspected only by function calls. This allows the implementation of a data type to be changed without requiring any changes outside the module of code in which that type is defined.

Abstract data types are often referred as *user-defined data types*, this is the way by which programmers can define their own data types. Just like a primitive type *Integer* with operations +, -, * , etc., an abstract data type has a type domain, whose representation is unknown to users, and a set of operations defined on the domain. Abstract Data Types were first formulated in their pure form in CLU [58, 55]. The theory of abstract data types is given by [61, 21]. Abstract Data Types are defined as

"entities that encapsulate information and provide operations to manipulate objects"

The users can only create values of an ADT by using the constructors, i.e. users of an ADT must use the well defined interface to act on the type.

First we give a somewhat lengthy presentation of the primitive data type *Integer*, which can be represented in a computer. Most of the programming languages, such as C, C++, `Java` and others, already offer an implementation for it. Sometimes it is called *int* or *integer*. Once created a variable of this type, we can use its provided operations. For instance, we can assign a value to a variable, we can add two integer numbers. We can define *Integer* ADT as:

Constructor: Creates an instance of an *Integer* ADT of type integer.

Operations: setValue, addValue etc.

Example 1.2.1 We discuss a very detailed implementation for the addition of two integer numbers:

int i,j,k; /* Define three integers */
i = 1; /* Assign 1 to integer i */
j = 2; /* Assign 2 to integer j */
k = i + j; /* Assign the sum of i and j to k */

In the above code, first line defines three instances i, j and k of type Integer. Consequently, for each instance the special operation constructor should be called, this is internally done by the compiler. The compiler reserves memory to hold the value of an integer and "binds" the corresponding name to it. If we refer to i, this actually refers to the memory area which was "constructed" by the definition of i. Optionally, compilers might choose to initialize the memory, for example, they might set it to 0 (zero).

The next line

i = 1;

sets the value of i to be 1. Therefore, it can be described with help of the ADT notation as:

Perform operation set with argument 1 on the Integer instance i. This is written as follows:

i.$setValue(1)$.

It has a representation at two levels. The first level is the ADT level where one expresses everything that is done to an instance of this ADT by the invocation of the defined operations. At this level, for complete specification, preconditions and postconditions are used to describe what actually happens. At this level, the conditions are mathematical conditions. In the example, these conditions are enclosed in curly brackets.

{ Precondition: i = n where n is any Integer }
i.$setValue(1)$
{ Postcondition: i = 1 }

The second level is the implementation level, where an actual *representation* is chosen for the operation. In C the equal sign "=" implements the $setValue()$ operation.

Let's stress these levels a little bit further and have a look at the line

k = i + j;

Obviously, "+" was chosen to implement the *addValue* operation and at the ADT level this results in

{ Precondition: Let i = n1 and j = n2 with n1, n2 particular Integers }
i.$addValue(j)$
{ Postcondition: i = n1 and j = n2 }

The postcondition ensures that i and j do not change their values. Specification of *addValue* says that a new Integer is created the value of which is the sum. The *setValue* operation is applied to access this new instance k:

{ Precondition: Let k = n where n is any Integer }
k.*setValue*(i.*addValue*(j))
{ Postcondition: k = i + j }.
However, the user cannot inspect the representation of integer values. ■

Example 1.2.2 Complex numbers cannot be represented natively in a computer. A Complex Number ADT can be defined as:

Constructor: Creates an instance of a Complex Number ADT using two *floats* (real, imaginary parts), a and b, where a represents the *real part* and b represents the *imaginary part*.

Operations: addition, subtraction, multiplication, division, comparison, etc.

■

For a complete specification, each such above operation of ADT should be defined with constraints i.e. inputs, outputs, preconditions, postconditions, and assumptions. ADTs are closely related to algebraic specifications [35, 37]. We discuss algebraic specifications further in Chapter 4. ADTs specify the meaning of operations (behaviour) *independently* of the concrete implementation. ADTs have the advantage of providing flexibility to modify an object implementation without affecting other routines, as long as the external interface remains the same.

Example 1.2.3 Consider the ADT List, for which functions are provided to create an empty list, to return the first element of the list, and to return the remaining elements of the list. The ADT List can be defined as:

Constructors: -for constructing an empty list, *nil*;
-for creating an instance of the ADT List, adding an element a to the front of a list.

Operations: head, tail, isEmpty.

■

Example 1.2.4 As another example, consider the ADT Stack, for which functions are provided to create an empty stack, to push values onto a stack, and to pop values from a stack. The ADT Stack can be defined as:

Constructors: -for constructing an empty stack, *nil*;
-for creating an instance of a stack using an element a to be added on top of a stack.

Operations: push, pop, top, isEmpty.

■

The number of ways a given ADT can be implemented depends on the programming language. For example, the stack example could be written in C using a struct and an accompanying set of data structures using arrays or linked lists to store the entries; however, the actual implementation is hidden from the user.

```
class Integer {
    attributes:
        int i
    methods:
        setValue(int n)
        Integer addValue(Integer j)
}
```

Figure 1.1: An example of class for the ADT *Integer*

1.3 Data Abstraction and Object Oriented Programming

Object-Oriented programming offers an alternative way to implement data abstraction besides ADTs. In OO-programming, data abstraction may refer to objects of a class, or to a special ADT created in traditional, non-OO-Programming languages. Therefore a class defines the properties of *objects* which are the instances in an object-oriented environment.

Data abstractions define functionality by putting main emphasis on the involved data, their structure, operations as well as axioms and preconditions. Consequently, data abstractions have a crucial role in OO-Programming. One can say that OO-Programming is programming with data abstraction *i.e.* combining functionality of different ADTs to solve the problem.

1.3.1 Data Abstraction as Classes

In object oriented programming classes are built around a hidden state space, which can only be observed and modified using certain specified operations. A user is not interested in the details of the actual implementation, but only in the behaviour that is realized.

A class is an actual representation of a data abstraction. It therefore provides implementation details for the data structure used and operations. We give an example of class for the ADT *Integer*, see Figure 1.1.

In Figure 1.1, **class** denotes the definition of a class, which consists of two sections, **attributes** and **methods**, which define the implementation of the data structure and the operations of the corresponding data abstraction. Again we distinguish the two levels with different terms. At the implementation level, we speak of attributes which are elements of the data structure. The same applies to methods which are the implementation of the operations.

In our example, the data structure consists of only one element: a signed sequence of digits. The corresponding attribute is an ordinary integer of a programming language. We only define two methods setValue() and addValue(), representing the two operations set and add.

A class introduces a collection of operations which are imperative, in the sense that methods modify objects which have an internal, updatable state.

In contrast, an ADT introduces a new type. Any operation on a value of the new type must take the value as an input parameter. With a pure ADT, there are no operations that modify values of the type. Instead, some operations can generate *fresh*, new values of the type.

1.3.2 ADT vs OO-Programming

Object-oriented programming involves the construction of objects which have a collection of methods, or procedures, that share access to private local state.

> Objects resemble machines or other things in the real world more than any well-known mathematical concept.
>
> -William R. Cook [24]

Abstract Data Types and Objects are two primary forms of data abstraction, but objects are not ADTs. The basic difference is in the mechanism used to achieve the abstraction barrier

	Constructor of s	
observations	nil	adjoin(s',n)
isEmpty?(s)	*true*	*false*
head(s)	**error**	n
tail(s)	**error**	s'

Figure 1.2: An observer/constructor specification for lists.

between a user and the data. Objects use *procedural* abstraction (methods), while ADTs use *type* abstraction -one that can be used by a user to declare variables but whose representation cannot be inspected directly. In *Object-Oriented programming*, the data is abstract because it is accessed through a procedural interface -although all of the types involved may be known to the user. This characterization is not completely strict, in that the type of a procedural data value can be viewed as being partially abstract, because not all of the interface may be known; in addition, abstract data types rely upon procedural abstraction for the definition of their operations.

Following William R. Cook [24] we distinguish ADTs and OO-Programming. As an example, consider a data abstraction for integer lists. The constructors are *nil*, which constructs an empty list, and *adjoin*, which takes a list and an integer, and forms a new list with the integer added to the front of the list argument. The observers are *isEmpty?, head,* and *tail*. *isEmpty?* is a predicate that returns true if its argument is the empty list; *i.e.* if it is equal to nil. *Head* returns the first integer in a non-empty list. *Tail* returns the rest of a non-empty list. The behavior of the observers on each constructor is given in Figure 1.2. The value of the *isEmpty?* observation on the *nil* constructor is *true*, and on the *adjoin* constructor it is false. The *head* and *tail* observations on *nil* both result an error condition. In Figure 1.2, *isEmpty?, head,* and *tail* observations are unary, since they observe a single value of the abstract data.

Abstract Data Types can be viewed as a decomposition of the specification matrix into observations, horizontal rows in Figure 1.2, where each row collects the information about a single observer together in a unit. Information about a given constructor is spread across components.

Each observation is formed into an independent operation that returns the appropriate result when applied to any of the constructors values. The constructors are also included as operations. The connection between the constructors and the observations is via a shared representation. In order to keep the representation abstract, its structure is hidden from users of the ADT. The users can only create values of the type by using the constructors, and inspect them only with observer operations. The concrete representation is usually derived from the form of the constructors, but alternative representations are also possible. Since they all share access to the real representation of the abstract type, the operations are tightly coupled.

An implementation of the ADT for integer lists is shown in Figure 1.3. The syntax is based loosely on ML. The ADT has two distinct parts: a representation and a set of operations.

(i) The *representation* is defined as a labeled union type, or variant record, with cases named NIL and CELL. The NIL variant is simply a constant, while the CELL variant contains a pair of an integer and a list. The constructors *nil* and *adjoin* are defined as operations that build appropriate representation values.

(ii) The *observations* are defined by a case statement over the representational variants which returns the appropriate value from the specification. *isEmpty*? is a query operation that determines if a list is nil. *head* and *tail* are accessors which return the first integer in the list, and the abstract list representing the tail, respectively.

```
adt IntList
representation
        list = NIL — CELL of integer * list
operations
        nil = NIL
        adjoin(x : integer, l : list) =
            CELL(x, l)
        isempty?(l : list) = case l of
            NIL⇒ true
            CELL(x, l)⇒ false
        head(l : list) = case l of
            NIL⇒ error
            CELL(x, l ) ⇒ x
        tail(l : list) = case l of
            NIL ⇒ error
            CELL(x, l ) ⇒ l
```

Figure 1.3: Implementation of an ADT for lists.

An user of the ADT is able to declare variables of type list and use the operations to create and manipulate list values.
var l : list = adjoin(adjoin(nil, 4), 7);

However, the user cannot inspect the representation of list values.

Procedural Data Abstraction (PDA) can be viewed as a decomposition of a data abstraction specification into constructors, columns in the Figure 1.2, each of which collects all the information about a given constructor into a unit. The values of the different observations are spread across the constructors.

Each constructor is converted into a class, for constructing procedural data values, or objects. The arguments to the constructor become the local state, or instance variables of the procedural data value. In the list example the *nil* object has no local state, while the *cons* object has two pieces of state: one to hold the integer value and the other to hold the rest of the list.

The observations become components or fields of the objects made by a constructor. The observations are often called attributes or methods. Each object is represented as the combination of the observations applicable to it. Since the result of an observation on a constructor is what a user is interested in, only the mechanism for producing the observed value needs to be hidden from the user.

In organizing the matrix in this way, a case discrimination is done on the operation to be performed, not on the constructor representations as in an ADT. Since operations are visible to the user, there is no need for a hidden case statement: the user can simply select the appropriate observation directly.

The two constructors for list objects are defined in Figure 1.4. The constructor functions, Nil and Cell, return record values. The constructor for cells takes two arguments, x and l, which play the role of instance variables of the object. In this example they are not changed by assignment, though there is no essential reason why they could not be modified (if, for example, a set-head method were introduced).

Each constructor creates a record with fields named *isEmpty?, head, tail,* and *cons*. The implementation uses recursive records, where the identifier self refers to the record being constructed. An observation m on an object o, which is written m(o) in the specification, is implemented by selecting the m field of the object: o.m. Explicit functional abstraction is introduced to represent the methods: the notation fun(x) represents a function of one argument named x. Unlike the ADT implementation, there are no explicit case statements in the PDAs. An implicit case statement is used to select the appropriate observation from the objects.

```
Nil = recursive self = record
      null? = true
      head = error;
      tail = error;
      cons = fun(y) Cell(y, self);
      end

Cell(x, l) = recursive self = record
      isEmpty? = false
      head = x;
      tail = l;
      cons = fun(y) Cell(y, self);
```

Figure 1.4: Implementation of lists as PDAs.

There are several levels of recursion in the implementation [**?**]. The Nil and Cell objects pass themselves as an argument to the Cell constructor, in response to a adjoin message. In addition, the Cell constructor function is itself recursive, because it is called from within the cell adjoin method. It is also possible for objects to return themselves as values of a method, as is common in Smalltalk. A user of the PDA creates objects and sends requests, or messages, to them.
var l = Nil.adjoin(4).adjoin(7);

As in the case of ADTs, the user cannot inspect the internal representation of list values, although the external format of all the messages are known.

Comparing ADT's and PDA's

We will briefly compare the pro's and con's of ADT's and PDA's.

- Adding new constructors in
  ADT: it is a little cumbersome, in that all operations (*i.e.* destructors) need to be modified;
  PDA: it is common practice and can be done without modifying any of the existing code.

- Adding new observations in
  ADT: it needs to access the hidden representation of the data types. Existing languages do not support it;
  PDA: it is common practice, it is the whole purpose of inheritance.

It is easier to optimize operations in ADT's than in PDA's. The same applies to formal verification. Part of the research in this thesis is a contribution to simplify the formal approach to PDA's.

In the light of the present thesis it will become clear that ADT's support amd implement an algebraic view of data abstraction while PDA support a coalgebraic view of data abstraction.

1.4 Inheritance

Inheritance is the mechanism which allows a class A to inherit properties of a class B. We say A *inherits from* B. Objects of class A thus have access to attributes and methods of class B without the need to redefine them.

The following definition defines two terms which refer classes related by inheritance.

Definition 1.4.1 (Superclass/Subclass) *If class A inherits from class B, then B is called superclass of A. A is called subclass of B. Objects of a subclass can be used where objects of the corresponding superclass are expected. This is due to the fact that objects of the subclass inherit the (same) behaviour of the objects of the superclass.*

```
class Point
   attributes
    xValue : int
    yValue : int
   methods
    pt.get-x = pt.xValue
    pt.get-y = pt.yValue
    pt1.eq(pt2 : Point) =
          if (pt1.get-x = pt2.get-x & pt1.get-y = pt2.get-y)
           then true
           else false
end class

class ColorPoint extends Point
   attributes
    sValue : int
   methods
    cpt.get-s = cpt.sValue
    cpt1.eq(cpt2 : Point) =
          if (cpt1.get-x = cpt2.get-x & cpt1.get-y = cpt2.get-y)
           then true
           else false
end class
```

Figure 1.5: An example of Inheritance with overriding functions

In object-oriented programming languages we also find other terms for *superclass* and *subclass*. Superclasses are also called parent classes. Subclasses may also be called child classes or just derived classes. For instance, see Figure 1.5, subclass *ColorPoint* inherits xValue and yValue variables and get-x, get-y methods from its superclass *Point*.

We can again inherit from a subclass, making this class the superclass of the new subclass. This leads to a hierarchy of superclass/subclass relationships. A subclass can use the methods of its superclass(es) or it can *override* them.

- *Overriding Methods.* In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, which implements a different version, or extends the existing version, of a superclass method, then the method in the subclass is said to *override* the method in the superclass. *E.g.* see Figure 1.5, method *eq* of class *Point* is overridden in class *ColorPoint* with same method name *eq* and same signature *Point*. Figure 1.6 shows another example, where method *total* of class *Seme1* is overridden in class *Seme2* with same name and signature, but, with different method bodies. When an *overridden* method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version in the superclass will be hidden.

- *Overloading Methods.* A method in a class is said to be *overloaded* if it has the same name and different type signatures as a method in its class (*e.g.* constructors of the class) or in its superclass. *E.g.* see Figure **??**, method *eq* of class *Point* is overloaded in class *ColorPoint* with same method name *eq* and different signature *ColorPoint*.

1.5 Types for Object-Oriented Languages

An interface type, also called an object type or simply a type contains the names of the object's methods, and the types of each method's arguments and results.

class *Seme1*
attributes
m1Value : *int*
m2Value : *int*
methods
stu.get-m1 = stu.m1Value
stu.get-m2 = stu.m2Value
stu.total = stu.m1Value+stu.m2Value
end class

class *Seme2* **extends** *Seme1*
atributes
m3Value : *int*
methods
stu.get-m3 = stu.m3Value
stu.total = stu.m1Value+stu.m2Value+stu.m3value
end class

Figure 1.6: Inheritance with overriding functions

Binary operations which take two arguments of the same type are quite familiar in non-object-oriented languages. Typical examples include arithmetic operations on number objects, as well as binary relations such as = and >, and set operations like subset and union. In object-oriented languages these operations are generally written as methods. In this case the first argument of the binary operation becomes the receiver of a corresponding "message", with the second parameter becoming the only argument. Consequently, we define a binary method of some object of type τ as a method that has an argument of the same type τ. Such a method is binary in the sense that it acts on two objects of the same type: the object passed as argument and the receiving object itself. In general, a binary method could also include other arguments (including other arguments of the same type); by a standard abuse of terminology we still refer to these as binary methods.

A subtype is a datatype that is generally related to another datatype (the supertype) by some notion of substitutability, meaning that computer programs written to operate on elements of the supertype can also operate on elements of the subtype. More specifically, the supertype-subtype relation is often taken to be the one defined by the Liskov substitution principle [56]; however, any given programming language may define its own notion of subtyping, or none at all.

Liskov substitution principle [57] formulated as

> *"Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T."*

Liskov and Wing's notion of *subtype* is based on the notion of substitutability; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).

In most class-based object-oriented languages, subclasses give rise to subtypes: if S is a subclass of T, then an instance of class S may be used in any context where an instance of type T is expected; thus we say S is a subtype of T. A consequence of this is that any variable or parameter declared as having type T might, at run time, hold a value of class S; in this situation many object-oriented programmers would say that T is the variable's static type and S is its dynamic type.

In general, the principle mandates that at all times objects from a class can be swapped with objects from an inheriting class, without the user noticing any other new behaviour. It has effects on the paradigms of design by contract, especially regarding to specification:

- postconditions for methods in the subclass should be more strict than those in the superclass;

class Point
  methods
    get-x : Self ⟶ real
    get-y : Self ⟶ real
    eq : Self × Self ⟶ bool
end Point

Figure 1.7: The class *Point*

class ColorPoint
  inherit from Point
  methods
    get-s : Self ⟶ string
    eq : Self × Self ⟶ bool
end ColorPoint

Figure 1.8: The class *ColorPoint*

- preconditions for methods in the subclass should be less strict than those in the superclass;
- no new exceptions should be introduced in the subclass.

For example, a language might allow integer values to be used wherever floating-point values are expected, or it might define a type number containing both the integers and the reals. In the first instance, the integer type would be a subtype of the floating-point type; in the second, those two types might be unrelated to each other by subtyping but both would be subtypes of number.

Typing binary methods is problematic when subclasses are considered as subtypes, *i.e.* a subclass object can always be passed to a method expecting a superclass object.

Binary methods when used with overloaded functions or methods in presence of inheritance have caused great difficulty for designers of object oriented languages and the programmers using those languages. Figure 1.7 shows the declaration of a class of points in a plane, a standard example of a class with a binary method. In a typical object oriented programming language all methods get an implicit first argument of type **Self**, to denote the type of the class that is being defined. The type of the first implicit argument is called **this** in C++, or **Current** in Eiffel. In the class *Point*, the method *get-x* returns the x-position of the plane, the method *get-y* returns the y-position and the method *eq* is binary. It takes two points as arguments to test for the equality and returns *true*, if they are considered as equal.

Figure 1.8 defines a subclass *ColorPoint* of the class *Point*. The implementation of method *eq* allowing two *ColorPoint* objects to be compared (taking the color into account), extends the code for method *equal* of *Point*.

In all OO-languages where subclasses are subtypes, the subclass *ColorPoint* can be used where an object of the superclass *Point* is expected. Unfortunately, *not* always subclasses produce subtypes. Assume that, we pass a colored point *cpt* into a method that expects an object of class *Point* and further assume that this method calls the method *eq* with a second argument *pt* of class *Point*. For instance, see Figure 1.9, we call the method *breakit* of class *Point* with an actual parameter colored point *cpt*; then, in this case the implementation of class *ColorPoint* for the method *eq* would be called. This code would try to access the color field of *pt*. Depending on the actual language used, this can yield anything between strange results, a runtime exception (possibly caught by the program), and a crash of the whole system. Because of the contravariance arrow type of subtyping relation on the domain of *eq*, *ColorPoint* is not a subtype of *Point*. This is not the case when we use inheritance with overriding (which is a mechanism to provide extended and consistent behaviour to subclasses) for *e.g.* see Figure 1.5, it will successfully executes the code for *equal* in *ColorPoint* with the second argument *pt* of class *Point* as well as with the second argument *cpt* of class *ColorPoint*. Therefore, the type of argument during inheritance is an important technique in object-oriented programming.

```
class Point
   attributes
    xValue : int
    yValue : int
   methods
    pt.get-x = pt.xValue
    pt.get-y = pt.yValue
    pt1.eq(pt2 : Point) =
          if (pt1.get-x = pt2.get-x & pt1.get-y = pt2.get-y)
           then true
           else false
    pt1.breakit(pt2 : Point) = if (pt2.eq(pt1))
                                 then true
                                 else false
end class

class ColorPoint extends Point
   attributes
    sValue : int
   methods
    cpt.get-s = cpt.sValue
    cpt1.eq(cpt2 : ColorPoint) =
          if (cpt1.get-x = cpt2.get-x & cpt1.get-y = cpt2.get-y &
                cpt1.get-s = cpt2.get-s)
           then true
           else false
end class
```

Figure 1.9: An example of Binary method with overloaded functions

In the last decades many researchers proposed semantic foundations for object-oriented programming on the basis of type theory. A large number of different proposals about how to solve the typing problem with binary methods are compared in [15].

One solution, which is for instance adopted by *OCaml*, [66, 65], is to separate subtyping and inheritance following the slogan Inheritance is not Subtyping from [25]. In this approach the typechecker would forbid the user to pass a colored point to a procedure that expects an ordinary point. However, this approach is quite restrictive. It denies many useful applications of argument type specialization.

Another interesting solution is proposed by Castagna in [19]. He suggests to enrich the basic calculus with multimethods and to use a more intelligent strategy for overriding and method dispatch. In the above example the method *equal* of class *ColorPoint* would have two implementations. The algorithm of *dynamic dispatch* would take both argument points into account (instead of only the first one) when choosing which implementation should be called, therefore it is also referred as *multiple dispatch*. For instance, when the method *equal* is called for the colored point *cpt* with an ordinary point *pt* as argument, the method *equal* of class *Point* would be called. The approach of Castagna is type safe. In Chapter 7 we shall discuss further the issue of typing binary methods in presence of inheritance, and we propose a somewhat simpler solution, albeit some what less general than Castagna's.

2

An Object-Oriented Language

In this chapter, we present the language Fickle [29, 30], as an example of `Java`-like OO-language. Fickle extends `Java` with *re-classification*, which allows objects to change class membership dynamically, while preserving their identities. We introduce various notions of Fickle's observational equivalences. In Chapter 6, we will study a coalgebraic semantics for Fickle and various notions of observational equivalences. In this chapter, we also introduce typing rules for safe Fickle expressions, classes and programs, in the line of [30]. In Chapter 7, we will discuss an alterative, more liberal typing system.

2.1 Syntax of Fickle

Fickle syntax is summarized in Table 2.1. A *Fickle program* P is a collection of (possibly *abstract*) class definitions. Fickle extends `Java` with re-classification. To this aim, a class definition may be preceded by the keyword **state** or **root**. State classes describe the properties of an object while it satisfies some conditions; when it no longer satisfies these conditions, it can be explicitly re-classified to another state class. Root classes abstract over state classes. While (state) classes consist of a sequence of fields and methods, in abstract (root) classes, some methods can only be declared. We assume the inheritance hierarchy to be a tree. Moreover, any subclass of a state or a root class must be a state class. Each state class must have a root (possibly indirect) superclass. Objects of a state class c may be re-classified to a class c', where c' must be a subclass of the uniquely defined root superclass of c. Objects of a non-state, non-root class c behave like `Java` objects, i.e. they are never re-classified. The type of fields may be either boolean or integer or a non-state class. Hence, fields cannot be reclassified. In contrast, the type of *this* and parameters may be a state or root class, i.e. these variables may be reclassified.

Objects are created with the expression **new** c, where c is any class. Re-classification expressions, $id \Downarrow c$, set the class of id to c, where c must be a state class.

Methods declarations have the shape:

$$t\ m\ (t_1x_1, \ldots, t_qx_q)\{c_1, \ldots, c_n\}\{\ e\ \}$$

where t is the result type, $t_1, \ldots, t_q$ are the types of the formal parameters $x_1, \ldots, x_q$ and e is the body. The list of root classes $c_1, \ldots, c_n$ are the effect, i.e. the root classes of all objects that may be re-classified by invocation of that method.

For simplicity, we assume all fields in the classes to be *private*, i.e. to be accessible from outside the class only through the class methods. On the contrary, we take all methods in a class to be *public*. Moreover, we assume no local variables in method bodies.

In Table 2.1, summarizing Fickle syntax, we have omitted the syntax of boolean and integer expressions, which involve the standard operators.

Example 2.1.1 (Lists in Fickle, [28]) The Fickle program below consists of three classes: the root class *List* (which is *abstract*, since it contains only a sequence of method declarations) together with two subclasses, *EmptyList* and *NonEmptyList*. This program uses re-classification, e.g. in the method *insertFront* of the class *EmptyList*. Some parts of the code are omitted.

progr	:=	class*	
class	:=	[**root** | **state**] **class** c **extends** c { field* meth* }	
absclass	:=	**abstract** [**root**] **class** c **extends** c { field* mdecl* }	
field	:=	type f	
meth	:=	type m (par*) eff { e }	
mdecl	:=	type m (par*) eff	
type	:=	**bool** | **int** | c	
par	:=	type x	
eff	:=	{ c* }	
expr (∋) e	:=	**if** e **then** e **else** e | var:=e | e;e | sVal | **this** | var | **new** c | e.m(e*) | id⇓c	
var	:=	x | e.f	
sVal	:=	**true** | **false** | **null** | 0 | 1 ...	
id	:=	**this** | x	
		with the following conventions	
c	::=	c | c' | c_i | d | ...	for class names
f	::=	f | f_i | ...	for field names
m	::=	m | m_i | m_{ij} | ...	for method names
x	::=	x | y | z | ...	for parameter names

Table 2.1: Syntax of Fickle

```
abstract root class List extends Objects{
      abstract insertFront(int i){List};
      abstract getFront(){List};
      abstract setFront(int i){List};
      abstract setLast(List x){ }; ...}

state class EmptyList extends List{
      void insertFront(int i){List}{
         this⇓NonEmptyList; contents:= i;
         next := new EmptyList; }
      int getFront (){}{ throw new ListException; } ...}

state class NonEmptyList extends List{
      int contents;
      List next;
      NonEmptyList insertFront(int i){}{
         NonEmptyList second:= new NonEmptyList;
         copyTo(second); contents:= i; next:=second; }
      int getFront(){List}{
         int result := contents; next.copyTo(this); return result;}
      List copyTo(NonEmptyList x){ } {
         x.contents := contents; x.next:=next; } ...}
```

■

2.2 Fickle's Operational Semantics

Operational semantics, a fundamental tool in language design and verification, provides a formal description of the behaviour of programs. It can be either defined as "small-step" operational semantics, *i.e.* in terms of atomic, elementary transitions, describing local behaviour, or as "big-step" operational semantics, *i.e.* providing final states for each configuration.

We describe the operational semantics of Fickle in terms of a "big-step" relation $\longrightarrow$, which rewrites pairs of expressions and stores w.r.t. to a program P into pairs of values, exceptions, or errors, and stores, representing the final state. The expression which is evaluated is meant to represent the special method *main* (external to P) from which the execution of the program starts. The type of the rewriting relation is:

$$\longrightarrow : \ \mathit{progr} \to \mathit{expr} \times \mathit{store} \to (\mathit{val} \cup \mathit{dev}) \times \mathit{store}$$

where:

$$\begin{array}{lcl}
\mathit{addr} & \triangleq & \mathit{Nat} \\
\mathit{val} & \triangleq & \mathit{sVal} \cup \mathit{addr} \\
\mathit{dev} & \triangleq & \{\ \mathit{nullPntrExc},\ \ \mathit{stuckErr}\ \} \\
\mathit{object}_c & \triangleq & \{[f_1 : v_1, \ldots, f_r : v_r]^c \mid\ f_1, \ldots, f_r \in \mathit{field}_c \text{ are} \\
 & & \quad \text{the field identifiers of c, } v_1, \ldots, v_r \in \mathit{val}\ \} \\
\mathit{object} & \triangleq & \bigcup_c \mathit{object}_c \\
\mathit{store} & \triangleq & (\{\mathit{this}\} \to \mathit{addr}) \times (\mathit{varid} \to_{\mathit{pfin}} \mathit{val}) \times (\mathit{addr} \to_{\mathit{pfin}} \mathit{object})\ ,
\end{array}$$

where *sVal* is defined in Table 2.1, *varid* is the set of variable identifiers, field_c is the set of field identifiers of c, and $\to_{\mathit{pfin}}$ denotes the space of partial functions with finite domain. Notice that an element of object_c is in fact a partial function in $(\mathit{field}_c \to_{\mathit{pfin}} \mathit{val})$.

In particular, stores are partial functions with *finite* domain, mapping *this* to an address, variables of base type to values, variables of class type to addresses, and addresses to objects. Notice in particular that, in the store, addresses point to objects, but *not* to other addresses. Thus in Fickle, as in `Java`, pointers are implicit, and there are no pointers to pointers. We denote addresses with ι, stores with σ, values with v, objects with o, exceptions and errors with dv.

Before introducing the rewriting rules, we need to define some operations on objects and stores. For object $o \triangleq [f_1 : v_1, \ldots, f_l : v_l, \ldots f_r : v_r]^c$, store σ, value v, address ι, identifier or address z, field identifier f, we define:

- *field access*: $o(f) \triangleq \begin{cases} v_l & \text{if } f = f_l \text{ for some } l \in 1, \ldots, r, \\ \mathit{Udf} & \text{otherwise} \end{cases}$
- *object update*: $o[v/f] \triangleq [f_1 : v_1, \ldots, f_l : v, \ldots f_r : v_r]^c$, where $f_l = f$ for some $l \in 1, \ldots, r$,
- *store update*: $\sigma[v/z](z) = v, \quad \sigma[v/z](z') = \sigma(z')$ if $z' \neq z$.

We use the convention that $\sigma(\iota)(f) = \mathit{Udf}$, whenever $\sigma(\iota) = \mathit{Udf}$, i.e. $\iota \notin \mathit{dom}(\sigma)$.

Tables 2.2, 2.3 and 2.4 list the rewriting rules of the operational semantics.

The evaluation of the expression **new** c in a store σ extends σ with a new *canonical address*. Moreover, all fields of the new object are initialized with canonical values, which we assume, by convention, to be **false** and 0 for boolean and integer fields, respectively, and **null** for fields of class type. The function $\mathcal{F}_S$ used in the rule for **new** (and for re-classification) is such that $\mathcal{F}_S(P, c)$ returns the set of fields defined in the class c, while $\mathcal{F}_S(P, c, f)$, used in the rule for reclassification, gives the type of the field f in class c.

In the rule for method call, $e_0.m(e_1, \ldots, e_n)$ in Table 2.2, we use the function $\mathcal{M}$: $\mathcal{M}(P, c, m)$ returns the definition of method m in class c going through the class hierarchy (see [30] for more details). Moreover, the premise $\sigma_n(\iota) = [...]^c$ means that, in the store σ'_n, the address ι refers to an object of the class c.

For re-classification expressions, $id \Downarrow d$, we find the address of id, which points to an object of class c. We replace the original object by a new object of class d. We preserve the fields belonging to the root superclass of c and initialize the other fields of d according to their types (as in the case of **new** expressions). The term $\mathcal{R}(P, t)$, defined by

$$\mathcal{R}(P, t) \triangleq \begin{cases} c & \text{if } t \text{ is a state class and } c \text{ is the root superclass of } t \\ t & \text{otherwise}\ , \end{cases}$$

denotes the least superclass of t which is not a state class, if t is a class, and denotes t itself if t is not a class.

2.3 Fickle's Observational Equivalences

Observational equivalences on programming languages are naturally induced by the operational semantics. Various notions of observational equivalence can be naturally introduced for Fickle. First of all, one can define a *contextual equivalence* on *main* methods w.r.t. a given program P, by evaluating the expressions corresponding to the bodies of the *main* methods in any expression context $C[\ ]$, and by observing the output value. A context is simply an expression with finitely many holes. As observable values, we take values of base types and errors/exceptions, i.e. $obsval \triangleq sVal \cup dev$. With $(e,\sigma) \Downarrow_P u$ we abbreviate the fact that there exists σ' such that $(e,\sigma) \rightarrow_P (u,\sigma')$, for $u \in sVal \cup dev$.

Definition 2.3.1 *(Contextual Equivalence):*
Let $\approx_P \subseteq expr \times expr$ *be defined by:* $e \approx_P e' \stackrel{\Delta}{\iff}$

$$\forall C[\ ]\ \forall \sigma\ \forall u \in obsval.\ (C[e],\sigma) \Downarrow_P u \ \Leftrightarrow\ (C[e'],\sigma) \Downarrow_P u\ .$$

The contextual equivalence $\approx_P$ on expressions e, e' induces an equivalence between a program P together with a *main* method whose body is the expression e, and the same program P together with a *main* method whose body is the expression e'. Notice that, by the assumption that all fields in a class are private (see Section 2.1), *main* methods can only access objects through class methods. In particular, in Definition 2.3.1 above, field access expressions appear neither in the expressions e, e' nor in the context $C[\ ]$.

In the definition of the observational equivalence $\approx_P$ above, the program P is fixed. However, in many cases, e.g. in program refinement, we are interested in establishing equivalences between different programs P_1, P_2, which implement the same program specification. A simple notion of program specification can be taken to be a list of *abstract classes* with no fields and only a sequence of method declarations. Then a program P_1 implements a program specification P, when the method declarations in each class of P_1 correspond exactly to the method declarations in P. One could consider a more sophisticated notion of program specification, involving a first-order logic for expressing conditions on the fields. This would be useful for studying program refinement (*e.g.* see [73]). By way of example, we introduce the following simple equivalence.

Two programs P_1, P_2, implementing the same program specification P, can be taken to be equivalent, when for any possible *main* method, they evaluate to the same value:

Definition 2.3.2 *Let* P_1, P_2 *implement the same program specification* P*. We define the equivalence* $\simeq$ *by:*

$$P_1 \simeq P_2 \ \stackrel{\Delta}{\iff}\ \forall e\ \forall u \in obsval.\ (e,\emptyset) \Downarrow_{P_1} u \ \Leftrightarrow\ (e,\emptyset) \Downarrow_{P_2} u\ .$$

The expression e in the above definition, being meant to represent a *main* method, is subject to the same syntactical restrictions as the expressions involved in Definition 2.3.1.

2.4 Typing Rules for Fickle programs

The following assertions, defined in Table 2.5, describe kinds of classes, and the widening relationship between types:

- $P \vdash c \diamond_{ct}$ means that c is any class,
- $P \vdash c \diamond_{rt}$ means that c is a re-classifiable type, *i.e.* , either a root or a state class,
- $P \vdash c \diamond_{ft}$ means that t is a field type, *i.e.* either bool or a non-state class,

$$\frac{(e,\sigma) \longrightarrow_P (\text{true},\sigma'') \quad (e_1,\sigma'') \longrightarrow_P (v,\sigma')}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (v,\sigma')} \qquad \frac{(e,\sigma) \longrightarrow_P (\text{false},\sigma'') \quad (e_2,\sigma'') \longrightarrow_P (v,\sigma')}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (v,\sigma')}$$

$$\frac{\sigma(x) \neq \mathit{Udf} \quad (e,\sigma) \longrightarrow_P (v,\sigma')}{(x := e,\sigma) \longrightarrow_P (v,\sigma'[v/x])} \qquad \frac{(e,\sigma) \longrightarrow_P (\iota,\sigma'') \quad (e',\sigma'') \longrightarrow_P (v,\sigma''') \quad \sigma'''(\iota)(f) \neq \mathit{Udf} \quad \sigma' \triangleq \sigma'''[\sigma'''(\iota)[v/f]/\iota]}{(e.f := e',\sigma) \longrightarrow_P (v,\sigma')}$$

$$\frac{(e_1,\sigma) \longrightarrow_P (v',\sigma'') \quad (e_2,\sigma'') \longrightarrow_P (v,\sigma')}{(e_1;e_2,\sigma) \longrightarrow_P (v,\sigma')} \qquad \frac{(e,\sigma) \longrightarrow_P (\iota,\sigma') \quad \sigma'(\iota)(f) \neq \mathit{Udf}}{(e.f,\sigma) \longrightarrow_P (\sigma'(\iota)(f),\sigma')}$$

$$\frac{\sigma(id) \neq \mathit{Udf}}{(id,\sigma) \longrightarrow_P (\sigma(id),\sigma)} \qquad \frac{}{(v,\sigma) \longrightarrow_P (v,\sigma)}$$

$$\frac{\mathcal{F}_S(P,c) = \{f_1,\ldots,f_r\} \quad v_l \text{ initial for } \mathcal{F}(P,c,f_l)\ (\forall l \in \{1,\ldots,r\}) \quad \iota \text{ is new in } \sigma}{(\textbf{new } c,\sigma) \longrightarrow_P (\iota,\sigma[[f_1 : v_1,\ldots,f_r : v_r]^c/\iota])}$$

$$\frac{(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0) \quad (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,n\}) \quad \sigma_n(\iota) = [\ldots]^c \quad \mathcal{M}(P,c,m) = t\ m(t_1x_1,\ldots,t_nx_n)\ \phi\ \{\ e\ \} \quad \sigma' = \sigma_n[\iota/\text{this}, v_1/x_1,\ldots,v_n/x_n] \quad (e,\sigma') \longrightarrow_P (v,\sigma'')}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (v,\sigma''[\text{this} \mapsto \sigma_n(\text{this}), x_1 \mapsto \sigma_n(x_1),\ldots,x_n \mapsto \sigma_n(x_n)])}$$

$$\frac{\sigma(id) = \iota \quad \sigma(\iota) = [\ldots]^c \quad \mathcal{F}_S(P,\mathcal{R}(P,c)) = \{f_1,\ldots,f_r\} \quad v_l = \sigma(\iota)(f_l)\ (\forall l \in \{1,\ldots r\}) \quad \mathcal{F}_S(P,d) \setminus \{f_1,\ldots,f_r\} = \{f_{r+1},\ldots,f_{r+q}\} \quad v_l \text{ initial for } \mathcal{F}_S(P,d,f_l)\ (\forall l \in \{r+1,\ldots r+q\})}{(id \Downarrow d,\sigma) \longrightarrow_P (\iota,\sigma[[f_1 : v_1,\ldots,f_{r+q} : v_{r+q}]^d/\iota])} \qquad \frac{(id,\sigma) \longrightarrow_P (\text{null},\sigma')}{(id \Downarrow d,\sigma) \longrightarrow_P (\text{null},\sigma')}$$

Table 2.2: Operational Semantics: execution without exceptions and errors

$$\frac{(e,\sigma) \longrightarrow_P (\text{null},\sigma')}{\begin{array}{l}(e.f := e',\sigma) \longrightarrow_P (\text{nullPntrExc},\sigma') \\ (e.f,\sigma) \longrightarrow_P (\text{nullPntrExc},\sigma') \\ (e.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\text{nullPntrExc},\sigma')\end{array}}$$

$$\frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (v,\sigma') \\ v \neq \text{true and } v \neq \text{false}\end{array}}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (\text{stuckErr},\sigma')}$$

$$\frac{\sigma(x) = \text{true or } \sigma(x) = \text{false}}{(x \Downarrow c,\sigma) \longrightarrow_P (\text{stuckErr},\sigma)}$$

$$\frac{\sigma(x) = \mathit{Udf}}{\begin{array}{l}(x,\sigma) \longrightarrow_P (\text{stuckErr},\sigma) \\ (x := e,\sigma) \longrightarrow_P (\text{stuckErr},\sigma) \\ (x \Downarrow c,\sigma) \longrightarrow_P (\text{stuckErr},\sigma)\end{array}} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (v,\sigma') \\ v \neq \text{null} \\ v \notin \text{addr}\end{array}}{\begin{array}{l}(e.f,\sigma) \longrightarrow_P (\text{stuckErr},\sigma') \\ (e.f := e',\sigma) \longrightarrow_P (\text{stuckErr},\sigma')\end{array}}$$

$$\frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma') \\ \sigma'(\iota)(f) = \mathit{Udf}\end{array}}{(e.f,\sigma) \longrightarrow_P (\text{stuckErr},\sigma')} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma'') \\ (e',\sigma'') \longrightarrow_P (v,\sigma') \\ \sigma'(\iota)(f) = \mathit{Udf}\end{array}}{(e.f := e',\sigma) \longrightarrow_P (\text{stuckErr},\sigma')}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (v,\sigma_0) \\ v \neq \text{null} \\ v \notin \text{addr or } \sigma_0(v) = \mathit{Udf}\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\text{stuckErr},\sigma_0)}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0) \\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,n\}) \\ \sigma_n(\iota) = [\ldots]^c \\ \mathcal{M}(P,c,m) = \mathit{Udf}\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\text{stuckErr},\sigma_n)}$$

Table 2.3: Operational semantics: generation of exceptions and errors

$$\frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\mathrm{dv},\sigma') \text{ or} \\ ((e,\sigma) \longrightarrow_P (\mathrm{true},\sigma'') \text{ and } (e_1,\sigma'') \longrightarrow_P (\mathrm{dv},\sigma')) \text{ or} \\ ((e,\sigma) \longrightarrow_P (\mathrm{false},\sigma'') \text{ and } (e_2,\sigma'') \longrightarrow_P (\mathrm{dv},\sigma'))\end{array}}{(\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}$$

$$\frac{(e_1,\sigma) \longrightarrow_P (\mathrm{dv},\sigma') \text{ or } ((e_1,\sigma) \longrightarrow_P (v,\sigma'') \text{ and } (e_2,\sigma'') \longrightarrow_P (\mathrm{dv},\sigma'))}{(e_1;e_2,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}$$

$$\frac{(e,\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}{\begin{array}{l}(x := e,\sigma) \longrightarrow_P (\mathrm{dv},\sigma') \\ (e.f,\sigma) \longrightarrow_P (\mathrm{dv},\sigma') \\ (e.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\mathrm{dv},\sigma') \\ (e.f := e',\sigma) \longrightarrow_P (\mathrm{dv},\sigma')\end{array}} \qquad \frac{\begin{array}{l}(e,\sigma) \longrightarrow_P (\iota,\sigma'') \\ (e',\sigma'') \longrightarrow_P (\mathrm{dv},\sigma')\end{array}}{(e.f := e',\sigma) \longrightarrow_P (\mathrm{dv},\sigma')}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0) \\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,q\},\ q < n) \\ (e_{q+1},\sigma_q) \longrightarrow_P (\mathrm{dv},\sigma_{q=1})\end{array}}{(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P (\mathrm{dv},\sigma_{q+1})}$$

$$\frac{\begin{array}{l}(e_0,\sigma) \longrightarrow_P (\iota,\sigma_0) \\ (e_i,\sigma_{i-1}) \longrightarrow_P (v_i,\sigma_i)\ (\forall i \in \{1,\ldots,n\}) \\ \sigma_n(\iota) = [\ldots]^c \\ \mathcal{M}(P,c,m) = t\ m(t_1x_1,\ldots,t_n : x_n)\ \phi\ \{\ e\ \} \\ \sigma' = \sigma_n[\mathrm{this} \mapsto \iota, x_1 \mapsto v_1,\ldots,x_n \mapsto v_n] \\ (e,\sigma') \longrightarrow_P (\mathrm{dv},\sigma'')\end{array}}{\begin{array}{l}(e_0.m(e_1,\ldots,e_n),\sigma) \longrightarrow_P \\ (\mathrm{dv},\sigma''[\mathrm{this} \mapsto \sigma_n(\mathrm{this}, x_1 \mapsto \sigma_n(x_1),\ldots,x_n \mapsto \sigma_n(x_n)])\end{array}}$$

Table 2.4: Operational semantics: propagation of exceptions and errors

$$\frac{\vdash P \diamond_u}{P \vdash O\text{bject} \sqsubseteq O\text{bject}} \qquad \frac{\vdash P \diamond_u \quad P = \ldots [\texttt{root} \mid \texttt{state}]\texttt{class c extends c}'\{\ldots\}\ldots}{P \vdash c \sqsubseteq c \quad P \vdash c \sqsubseteq c'}$$

$$\frac{P \vdash c \sqsubseteq c' \quad P \vdash c' \sqsubseteq c''}{P \vdash c \sqsubseteq c''}$$

$$\frac{\begin{array}{l}\forall c, c' : \\ P \vdash c \sqsubseteq c' \text{ and } P \vdash c' \sqsubseteq c \implies c = c' \\ \quad \mathcal{C}(P, c) = \texttt{class c extends c}'\ \{\ldots\} \implies \mathcal{C}(\texttt{P}, \texttt{c}') = \texttt{class c}'\ \ldots \\ \quad \mathcal{C}(P, c) = \texttt{root class c extends c}'\ \{\ldots\} \implies \mathcal{C}(\texttt{P}, \texttt{c}') = \texttt{class c}'\ \ldots \\ \quad \mathcal{C}(P, c) = \texttt{state class c extends c}'\ \{\ldots\} \implies \\ \qquad ((\mathcal{C}(P, c') = \texttt{root class c}' \ldots) \text{ or } (\mathcal{C}(\texttt{P}, \texttt{c}') = \texttt{state class c}'\ \ldots))\end{array}}{\vdash P \diamond_h}$$

$$\frac{\vdash P \diamond_h \quad \mathcal{C}(P, c) = \texttt{class c}\ \ldots}{P \vdash c \diamond_{ft}} \qquad \frac{\vdash P \diamond_h \quad \mathcal{C}(P, c) = \texttt{root class c}\ \ldots}{P \vdash c \diamond_{ft} \quad P \vdash c \diamond_{rt} \quad P \vdash c \diamond_{ct}}$$

$$\frac{\vdash P \diamond_h \quad \mathcal{C}(P, c) = \texttt{state class c}\ \ldots}{P \vdash c \diamond_{rt} \quad P \vdash c \diamond_{ct}}$$

$$\frac{}{P \vdash \mathit{bool} \diamond_{ft}} \qquad \frac{}{P \vdash \mathit{bool} \leq \mathit{bool}} \qquad \frac{P \vdash c \sqsubseteq c'}{P \vdash c \leq c'}$$

Table 2.5: Subclasses, well-formed inheritance hierarchy, subtypes

- $P \vdash t \leq t'$ means that type t' widens type t, *i.e.* t is a subclass of, or identical to, t'.

Environments, Γ, map parameter names to types, and the receiver *this* to a class. They have the form $\{x_1 : t_1, \ldots, x_n : t_n, this : c\}$. Lookup, $\Gamma(id)$, and update, $\Gamma[id \mapsto t]$, have the usual meaning, (see [30] for more details).

Typing an expression e in the context of a program P and an environment Γ involves three components, namely

$$P, \Gamma \vdash e : t \llbracket \Gamma' \rrbracket \phi$$

where t is the type of the value returned by evaluation of e, the environment Γ' contains the type of *this* and of the parameters after evaluation of e, and ϕ conservatively estimates the re-classification effect of the evaluation of e on objects.

The typing rules are given in Table 2.6. We use the look-up functions $\mathcal{F}(P, c, f)$ and $\mathcal{M}(P, c, m)$, (see [30] for more details), which search for fields and methods through the class hierarchy. We follow the convention that rules can be applied only if the types in the conclusion are defined. This is useful in rules (*cond*) and (*id*).

Consider the rule (*seq*) for composition e, e'. The second expression, e', is typed in the environment Γ', *i.e.* the environment updated by typing the first expression, e. The effect of the composition is the union of the effects of the components.

Consider now the rule (*cond*) for conditionals. With $t \sqcup_P t'$ we denote the least upper bound of t and t' in P with respect to $\leq$, when it exists[1]. With $\Gamma \sqcup_P \Gamma'$ we denote the extension of the

[1]Note that for any class c the least upper bound $c \sqcup_P \mathit{bool}$ does not exist

cond

$$\frac{P,\Gamma \vdash e : bool \parallel \Gamma_0 \parallel \phi \quad P,\Gamma_0 \vdash e_1 : t_1 \parallel \Gamma_1 \parallel \phi_1 \quad P,\Gamma_0 \vdash e_2 : t_2 \parallel \Gamma_2 \parallel \phi_2}{P,\Gamma \vdash (\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2) : t_1 \sqcup_P t_2 \parallel \Gamma_1 \sqcup_P \Gamma_2 \parallel \phi \cup \phi_1 \cup \phi_2}$$

a-field

$$\frac{P,\Gamma \vdash e : c \parallel \Gamma_0 \parallel \phi \quad P,\Gamma_0 \vdash e' : t \parallel \Gamma' \parallel \phi' \quad \mathcal{F}(P, \phi'@_p c, f) = t' \quad P \vdash t \leq t'}{P,\Gamma \vdash e.f := e' : t \parallel \Gamma' \parallel \phi \cup \phi'}$$

a-var

$$\frac{P,\Gamma \vdash e : t' \parallel \Gamma' \parallel \phi \quad \Gamma'(x) = t \quad P \vdash t' \leq t}{P,\Gamma \vdash x := e : t' \parallel \Gamma' \parallel \phi}$$

field

$$\frac{P,\Gamma \vdash e : c \parallel \Gamma' \parallel \phi \quad \mathcal{F}(P, c, f) = t}{P,\Gamma \vdash e.f : t \parallel \Gamma' \parallel \phi}$$

seq

$$\frac{P,\Gamma \vdash e : t \parallel \Gamma_0 \parallel \phi \quad P,\Gamma_0 \vdash e' : t' \parallel \Gamma' \parallel \phi'}{P,\Gamma \vdash e; e' : t' \parallel \Gamma' \parallel \phi \cup \phi'}$$

bool

$$\frac{}{P,\Gamma \vdash true : bool \parallel \Gamma \parallel \{\,\}} \qquad \frac{}{P,\Gamma \vdash false : bool \parallel \Gamma \parallel \{\,\}}$$

null

$$\frac{P \vdash c \diamond_{ct}}{P,\Gamma \vdash null : bool \parallel \Gamma \parallel \{\,\}}$$

id

$$\frac{}{P,\Gamma \vdash id : \Gamma(id) \parallel \Gamma \parallel \{\,\}}$$

new

$$\frac{P \vdash c \diamond_{ct}}{P,\Gamma \vdash \textbf{new } c : c \parallel \Gamma_0 \parallel \{\,\}}$$

meth

$$\frac{P,\Gamma \vdash e_0 : c \parallel \Gamma_0 \parallel \phi_0 \quad P,\Gamma_{i-1} \vdash e_i : t'_i \parallel \Gamma_i \parallel \phi_i \ (\forall i \in \{1,\ldots,n\}) \quad \mathcal{M}(P, (\phi_1 \cup \ldots \cup \phi_n)@_p c, m) = t\ m(t_1 x_1, \ldots, t_n x_n)\ \phi\ \{\ldots\} \quad P \vdash (\phi_{i+1} \cup \ldots \cup \phi_n)@_p t'_i \leq t_i \ (\forall i \in \{1,\ldots,n\})}{P,\Gamma \vdash (e_0.m(e_1,\ldots,e_n)) : t \parallel \phi@_p\Gamma_n \parallel \phi \cup \phi_0 \cup \ldots \cup \phi_n}$$

recl

$$\frac{P \vdash c \diamond_{ct} \quad \mathcal{R}(P,c) = \mathcal{R}(P, \Gamma(id))}{P,\Gamma \vdash id \Downarrow c : c \parallel (\{\mathcal{R}(P,c)\}@_p\Gamma)[id \mapsto c] \parallel \{\mathcal{R}(P,c)\}}$$

Table 2.6: Typing rules for *Fickle* expressions

$$\frac{\begin{array}{l}\mathcal{C}(P,c) = [\texttt{root} \mid \texttt{state}]\ \texttt{class c extends c}'\ \{\ldots\} \\ \forall f:\ \mathcal{FD}(P,c,f) = t_0 \implies P \vdash t_0 \diamond_{ft} \text{ and } \mathcal{F}(P,c',f) = Udf \\ \forall m:\ \mathcal{MD}(P,c,m) = t\ m(t_1x_1,\ldots,t_nx_n)\ \phi\ \{\ e\}\ \implies \\ \quad P \vdash \phi \diamond \\ \quad P, \{x_1 : t_1, \ldots, x_n : t_n, this; c\} \vdash e : t' \parallel \Gamma' \parallel \phi' \\ \quad P \vdash t' \leq t \\ \quad \phi' \subseteq \phi \\ \quad \mathcal{M}(P,c',m) = \mathcal{U}df \text{ or} \\ \qquad (\mathcal{M}(P,c',m) = t\ m(t_1x_1,\ldots,t_nx_n)\ \phi\ \{\ \ldots\ \} \text{ and } \phi \subseteq \phi'')\end{array}}{P \vdash c \diamond}$$

$$\frac{\vdash P \diamond_h \qquad \forall c;\ \mathcal{C}(P,c) \neq Udf \implies P \vdash c \diamond}{\vdash P \diamond}$$

Table 2.7: Typing rules for *Fickle* classes and programs

above operation to environments, defined as follows:

$$\Gamma \sqcup_P \Gamma' = \{id : (t \sqcup_P t') \mid \Gamma(id) = t \text{ and } \Gamma'(id) = t'\}$$

Least upper bounds are used in rule (*cond*) to determine a conservative approximation of the type of the conditional expression. The two branches may cause different re-classifications for this and the parameters. So, after the evaluation we can only assert that *this* and the parameters belong to the least upper bound of their relative classes in Γ_1 and Γ_2.

Consider now the typing of assignments, *i.e.* rules (*a-field*) and (*a-var*). Evaluation of the right hand side may modify the type of the left hand side. In particular, in (*a-var*) evaluation of e can modify the type of x. This is taken into account by looking up x in the environment Γ'. Also, in rule (*a-field*) evaluation of e' may modify the class of the object e. For this purpose, we define the application of effects to types:

$$\{c_1, \ldots, c_n\}@_P t = \begin{cases} c_i & \text{if } \mathcal{R}(P,t) = c_i \text{ for some } i \in 1,\ldots,n \\ t & \text{otherwise.} \end{cases}$$

Consider now (*recl*) : $id \Downarrow c$ is type correct if c, the target of the re-classification, is a state or root class, and if c and the class of id before the re-classification (the class $\Gamma(id)$) are subclasses of the same root class. A re-classification updates the environment by changing the class of the identifier id. Moreover, since there could be aliasing with identifiers of state classes that are subclasses of the root class of id, the static type of all such variables is set to the root class. For this reason, we define the application of effects to environments:

$$\phi@_P\Gamma = \{id : \phi@_P t \mid \Gamma(id) = t\}$$

Consider rule (*meth*) for method calls, $e'.m((e_1,\ldots,e_n))$. The evaluation of the arguments $e_{i+1},\ldots,e_n$ may modify the types of the arguments $e_1,\ldots,e_i$ and of the object e'. This could happen if a superclass of the original type of $e_j (1 \leq j \leq i)$ is among the effects of $e_{i+1},\ldots,e_n$. (Existence of such a class implies uniqueness, since effects are sets of root classes.) The definition of m has to be found in the new class of the object e_0, and the types of the formal parameters must be compared with the new types of $e_1,\ldots,e_{n-1}$. In (*meth*) we look up the definition of m in the class obtained by applying the effect of the arguments to the class of the receiver and we compare the types of formal and actual parameters by keeping into account the effects of the actual parameters.

A program is well formed (*i.e.* $\vdash P \diamond$) if the inheritance hierarchy is well-formed (*i.e.* $\vdash P \diamond_h$) and all its classes are well-formed (*i.e.* $P \vdash c \diamond$). Fields may not redefine fields from superclasses, and methods may redefine superclass methods only if they have the same name, arguments, and result type, and their effect is a subset of that of the overridden method. Method bodies must be well formed, must return a value appropriate for the method signature, and their effect must be a subset of that in the signature. See Table 2.7, where $\mathcal{C}(P, c)$ returns the definition of class c in program P, and the look-up functions $\mathcal{FD}(P, c, f)$, $\mathcal{MD}(P, c, m)$, (see [30] for more details), search for fields and methods only in class c.

The judgment $P, \sigma \vdash v \triangleleft t$, guarantees that value v conforms to type t. In particular, it requires that an address ι points to an object of class c, a subclass of t, that the object contains all fields required in the description of c, and that the fields contain values which conform to their type in c. The judgment $P, \Gamma \vdash \sigma \diamond$ guarantees that all object fields contain values which conform to their types in the class of the objects, and that all parameters and the receiver are mapped to values which conform to their types in Γ. For formal definitions see [30].

The type system is sound in the sense that a converging well-typed expression returns a value which agrees with the expression's type, or `nullpntrExc`; but it is *never* stuck.

Theorem 2.4.1 *[27] For a well-formed program P, environment Γ, and expression e, such that*

$$P, \Gamma \vdash e \ : \ t[\![\Gamma']\!]\phi$$

if $P, \Gamma \vdash \sigma\diamond$, and e, σ converges then

- $e,\ \sigma \ \longrightarrow_P \ v,\ \sigma', \quad P,\ \sigma' \vdash v \triangleleft\ t, \quad P, \Gamma' \vdash \sigma'\diamond,$

  or

- $e,\ \sigma \ \longrightarrow_P \ \mathit{nullpntrExc},\ \sigma'.$

II

Second Part : The Theory of Co(bi)algebras

3

Bialgebraic Preliminaries

In this chapter, we describe the basic notions and results about algebras, coalgebras, bialgebras. We also discuss congruences, bisimulation and bicongruences. Throughout the thesis, $\mathcal{C}$ denotes a *category*, and $F, G, H, L : \mathcal{C} \to \mathcal{C}$ denote endofunctors on $\mathcal{C}$.

3.1 Algebras

In this section, following [50], we introduce the basic notions of *algebra*, *algebra morphism* and *initial algebra*.

An algebra is a set plus some operations on it, and as such is used to describe many kinds of data types in programming languages, like stacks, lists. For example, let us consider the set of the integer numbers Z, with *sum* (a binary operation), *negation* (a unary operation) and the number *zero* (a constant, i.e., a zero-ary operation). This is an algebra. The signature of an algebra is the information relative to the arity of its operations, their names being influent.

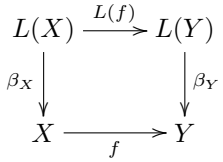
Let us indicate

- $A \times B$ the cartesian product of two sets A and B, i.e., the sets of all the ordered pairs of elements drawn one from each set;
- $A \times A$ with the simpler notation A^2;
- 1 the singleton set, $1 = \{*\}$;
- disjoint union of the sets A and B with $A + B$.

Now, the three operations of our algebra can be seen as a single bundle operation from $Z^2 + Z + 1$ to Z. The signature of our algebra can be written as $[\,]^2 + [\,] + 1$, where $[\,]$ is a placeholder for the carrier of choice, Z.

If we consider a generic signature L, an algebra with signature L and carrier X is any function from $L(X)$ to X, i.e., any function picking a complexly structured value, among whose subparts there may be values from X, and returning a value from X. Therefore, an algebra consists of a carrier set with certain functions called *constructors*, since they represent operations which *build* elements of the carrier. For a given signature L, the *algebra homomorphisms* are those functions between two L-algebras which preserve the L-structure. Categorically:

Definition 3.1.1 (L-algebra) *Let $L : \mathcal{C} \to \mathcal{C}$. An* L-algebra *is a pair (X, β_X), where X is an object of $\mathcal{C}$, also named "carrier", and $\beta_X : L(X) \to X$ is an arrow of $\mathcal{C}$, also named "structure" or "operation". L-algebras can be endowed with the structure of a category, Alg_L, by defining L-algebra morphisms as follows. $f : (X, \beta_X) \to (Y, \beta_Y)$ is an* L-algebra morphism *if $f : X \to Y$ is an arrow of the category $\mathcal{C}$ such that the following diagram commutes*



Here are a couple of standard examples, see *e.g.* [50] for others.

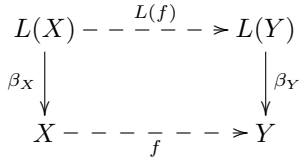
Example 3.1.2 The set of A-labeled lists $List(A)$ comes with functions $nil : 1 \to List(A)$ for the empty list and $cons : A \times List(A) \to List(A)$ for constructing a list of type A. By joining the two constructors to a single one, we form an algebra $[nil, cons] : 1 + (A \times List(A)) \to List(A)$ of the functor $L(X) = 1 + (A \times X)$. ■

Example 3.1.3 The set of A-labeled finite binary trees $Tree(A)$ comes with functions $nil : 1 \to Tree(A)$ for the empty tree, and $node : Tree(A) \times A \times Tree(A) \to Tree(A)$ for constructing a tree out of two subtrees and a (node) label. Together, nil and $node$ form an algebra $1 + (Tree(A) \times A \times Tree(A)) \to Tree(A)$ of the functor $L(X) = 1 + (X \times A \times X)$. ■

3.1.1 Initial Algebras

An algebra is initial if there is *exactly one* homomorphism from it to any other algebra (with the same signature, of course). That is, every algebra contains an *image* of an initial algebra.

Definition 3.1.4 (Initial L-algebra) *An* algebra $\beta_X : L(X) \to X$ *of a functor* L *is* initial *if for each* algebra $\beta_Y : L(Y) \to Y$ *there is a* unique *homomorphism of algebras* $f : (X, \beta_X) \to (Y, \beta_Y)$*:*



A familiar example of initial algebra is the initial algebra of the functor $1 + [\ ]$, which is the algebra of natural numbers with zero and successor. This has all the properties of Peano's definition. Namely, Initiality implies that the operation $\langle 0, S \rangle : 1 + N \to N$ is an isomorphism, i.e., that the natural numbers are "as many as" (i.e. isomorphic to) the natural numbers plus another element, the zero, and that the successor operation S is a bijection from naturals without zero. This fixes the first four Peano axioms. Minimality yields the induction principle, which is the fifth axiom.

Both A-labeled lists and A-labeled trees of Examples 3.1.2 and 3.1.3 are *initial* algebras for the respective functors.

A natural question which arises is when does an initial algebra exist? In case of *class-theoretic categories*, *i.e.* categories whose objects are classes and whose maps are functions between classes, the following strong result holds:

Theorem 3.1.5 (Initial Algebra Theorem) *Every endofunctor on a class-theoretic category has an initial algebra.*

The above theorem is a refinement of a standard theorem, (see [5] "Lectures on semantics : The initial algebra and final coalgebra prespectives") which has been recently obtained by [8, 17]

3.2 Coalgebras

In this section, following [50], we introduce the basic notions of *coalgebra*, *coalgebra morphism*, and *final coalgebra*.

A coalgebra is the dual of an algebra. As algebras are used to formalize data types, coalgebras are used to formalize automata and similar computational systems, or assimilable data types, like streams, which may be infinite or circular. For instance, let us consider a finite state automaton with states in the set S, with I as input alphabet and with O as output alphabet. An automaton is defined by its transition function, taking a pair (state, input) and returning a pair (new state, output). In a stream, by a function from $S \times I$ to $S \times O$. If we apply currying, any function of two arguments can be conveniently described by an equivalent function consuming its first argument, and returning another function which consumes the second one. This means, any function from $S \times I$ to $S \times O$ can be conveniently described by an equivalent function from S to $(S \times O)^I$, where $(S \times O)^I$ is the set of all the functions from I to $S \times O$. Thus, an automaton with input alphabet I and output alphabet O is a coalgebra with signature $([\,] \times O)^I$, whose carrier is the set of its states.

In general, given a signature H, a coalgebra with signature H and carrier X is any function from X to $H(X)$, i.e., any function picking a value from X and returning a complexly structured value, among whose subparts there may be values from X. Therefore, a coalgebra consists of a carrier set with certain operations going *out* of the carrier set. These are good for describing *destructors* or *observers*, which allows us to observe certain behaviours. Categorically:

Definition 3.2.1 (H-coalgebra) *Let $H : \mathcal{C} \to \mathcal{C}$. A H-*coalgebra *is a pair (X, α_X), where X is an object of $\mathcal{C}$, also named "state space", and $\alpha_X : X \to H(X)$ is an arrow of $\mathcal{C}$, also named "structure" or "operation" of the H-coalgebra (X, α_X). H-coalgebras can be endowed with the structure of a category, $Coalg_H$, by defining H-coalgebra morphisms as follows. $f : (X, \alpha_X) \to (Y, \alpha_Y)$ is an H-*coalgebra morphism *if $f : X \to Y$ is an arrow of the category $\mathcal{C}$ such that the following diagram commutes*

$$\begin{array}{ccc} X & \xrightarrow{\ f\ } & Y \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ H(X) & \xrightarrow[H(f)]{} & H(Y) \end{array}$$

Example 3.2.2 The set of infinite A-labeled streams, $String(A)$, form a coalgebra for the functor $H(X) = A \times X$. This is given by $(\, String(A), \alpha : String(A) \to H(String(A))\,)$ where $\alpha = \langle \alpha_{head}, \alpha_{tail} \rangle$ consists of two functions $\alpha_{head} : String(A) \to A$, $\alpha_{tail} : String(A) \to String(A)$, where α_{head} yields the first element of an infinite sequence of elements of A, and α_{tail} yields the remaining string.

■

3.2.1 Final Coalgebras

A coalgebra is final if there is *exactly one* homomorphism from any other coalgebra (again, with the same signature). Dually to initial algebras, final coalgebras contain the *image* of every coalgebra.

Definition 3.2.3 (Final H-coalgebra) *A* coalgebra $\alpha_X : X \to H(X)$ *of a functor H is* final *if for each* coalgebra $\alpha_Y : Y \to H(Y)$ *there is a* unique *homomorphism of coalgebras, $f : (Y, \alpha_Y) \to (X, \alpha_X)$:*

$$\begin{array}{ccc} Y & \overset{f}{\dashrightarrow} & X \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ H(Y) & \underset{H(f)}{\dashrightarrow} & H(X) \end{array}$$

An example of final coalgebra is the familiar coalgebra of streams, see Example 3.2.2.

The following is a very strong result about the existence of final coalgebras of class-theoretic endofunctors. This strengthens earlier results of Aczel and Mendler [3, 6] and Adamek et al. eds, [7, 8]. This result has been recently obtained independently by [9] and [17].

Theorem 3.2.4 (Final Coalgebra Theorem) *Every endofunctor on a class-theoretic category has a final coalgebra.*

3.3 Bialgebras

In this section, we introduce the basic notions of *bialgebra*, *bialgebra morphism*, λ-*bialgebra*, and *final bialgebra*. We start by introducing the notion of $\langle L, H\rangle$-*bialgebra*, which combines the structures of L-algebra and H-coalgebra independently:

Definition 3.3.1 ($\langle L, H\rangle$-bialgebra) *Let $L, H : \mathcal{C} \to \mathcal{C}$. An* $\langle L, H\rangle$-bialgebra *is a triple (X, β_X, α_X), where X is an object of $\mathcal{C}$, (X, β_X) is an L-algebra, and (X, α_X) is an H-coalgebra. $\langle L, H\rangle$-bialgebras can be endowed with the structure of a category, $Bialg_{L,H}$, by defining $\langle L, H\rangle$-bialgebra morphisms as follows. $f : (X, \beta_X, \alpha_X) \to (Y, \beta_Y, \alpha_Y)$ is an* $\langle L, H\rangle$-bialgebra morphism *if $f : X \to Y$ is an arrow of the category $\mathcal{C}$ such that the following diagram commutes*

$$\begin{array}{ccccc} L(X) & \xrightarrow{\beta_X} & X & \xrightarrow{\alpha_X} & H(X) \\ {\scriptstyle L(f)}\downarrow & & \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle H(f)} \\ L(Y) & \xrightarrow[\beta_Y]{} & Y & \xrightarrow[\alpha_Y]{} & H(Y) \end{array}$$

That is, the morphisms $f : (X, \beta_X, \alpha_X) \to (Y, \beta_Y, \alpha_Y)$ of $Bialg_{L,H}$ are those morphisms $f : X \to Y$ between the carriers which are both L-algebra and H-coalgebra homomorphisms.

In the above definition the two structures of L-algebra and H-coalgebra are combined independently. More interesting is the case where there is a connection between the two structures. λ-bialgebras [78] are defined as algebra-coalgebra pairs over a common carrier X subjected to a *pentagonal law*, which ensures that the same structure can be seen both as a coalgebra in the category of L-algebras and as an algebra in the category of H-coalgebras. The notion of λ-bialgebra is given in terms of a *distributive law*.

Definition 3.3.2 (Distributive law[12]) *Let $L, H : \mathcal{C} \to \mathcal{C}$. A distributive law of L over H is a natural transformation (see Appendix 7.3) $\lambda : LH \dot{\Rightarrow} HL$.*

Definition 3.3.3 (λ-bialgebra) *Let $L, H : \mathcal{C} \to \mathcal{C}$. A λ-bialgebra, for a given distributive law $\lambda : LH \dot{\Rightarrow} HL$, is an $\langle L, H\rangle$-bialgebra (X, β_X, α_X) which satisfies the following* pentagonal law*:*

$$\alpha_X \circ \beta_X = H(\beta_X) \circ \lambda_X \circ L(\alpha_X)$$

$$\begin{array}{ccccc} LHX & & \xrightarrow{\lambda_X} & & HLX \\ {\scriptstyle L(\alpha_X)}\uparrow & & & & \downarrow{\scriptstyle H(\beta_X)} \\ L(X) & \xrightarrow[\beta_X]{} & X & \xrightarrow[\alpha_X]{} & H(X) \end{array}$$

In the above definition the *pentagonal law* makes α_X an L-algebra homomorphism and β_X an H-coalgebra homomorphism, respectively. λ-bialgebras form a subcategory of $\langle L, H\rangle$-bialgebra, Definition 3.3.3 above of λ-bialgebra corresponds to that introduced in [78], apart from the fact that we are considering λ-bialgebras simply for endofunctors and we are not assuming that L and H are monads and comonads as in [78].

The notion of λ-bialgebra, as we have given, corresponds essentially, to the notion of *structured coalgebra* of [26, 27]. *Structured coalgebras* are simply defined as coalgebras on a category of algebras, formally:

Definition 3.3.4 (Structured Coalgebra) *Let $L : \mathcal{C} \to \mathcal{C}$ be a functor and let H^+ be a functor on the category of L-algebras. A* structured coalgebra *is an H^+-coalgebra.*

In order to state formally the precise correspondence between λ-bialgebras and structured coalgebras, we need to introduce the following definition:

Definition 3.3.5 (Functor Lifting) *Let $F : \mathcal{C} \to \mathcal{C}$, $F^+ : \mathcal{C}' \to \mathcal{C}'$, $V : \mathcal{C}' \to \mathcal{C}$ be endofunctors; F^+ is called lifting of F along V if $V \circ F^+ = F \circ V$.*

Theorem 3.3.6 *Let $L, H : \mathcal{C} \to \mathcal{C}$, and let $\lambda : LH \stackrel{\cdot}{\Rightarrow} HL$ be a distributive law, then there exists both a*

1. *lifting L^+ of L to the category of H-coalgebras;*
2. *lifting H^+ of H to the category of L-algebras.*

Proof. 1. Let $\lambda : LH \stackrel{\cdot}{\Rightarrow} HL$ be a distributive law. We define $L^+ : Coalg_H \to Coalg_H$ as follows:

For any H-coalgebra (X, α_X), $L^+(X, \alpha_X) = (LX, \alpha_{LX})$, where $\alpha_{LX} : LX \to H(LX)$ is defined by

$$\alpha_{LX} \triangleq \lambda_X \circ L\alpha_X$$

For any $f : (X, \alpha_X) \to (Y, \alpha_Y)$ H-coalgebra morphism, we define $L^+(f)$ as $L(f) : LX \to LY$. $L^+(f)$ is a coalgebra morphism from $(HX, \lambda_X \circ L\alpha_X)$ to $(HY, \lambda_Y \circ L\alpha_Y)$, since the following diagrams commute:

$$\begin{array}{ccccc} LX & \xrightarrow{L\alpha_X} & LHX & \xrightarrow{\lambda_X} & HLX \\ {\scriptstyle L(f)}\downarrow & (1) & {\scriptstyle LH(f)}\downarrow & (2) & \downarrow{\scriptstyle HL(f)} \\ LY & \xrightarrow[L\alpha_Y]{} & LHY & \xrightarrow[\lambda_Y]{} & HLY \end{array}$$

One can easily check that L^+ is a functor on the category of H-coalgebras. In order to show L^+ is a lifting of L, we need to show that $V \circ L^+ = L \circ V$, for a functor $V : Coalg_H \to \mathcal{C}$. This holds if we consider V to be the forgetful functor (see Appendix 7.3).

2. Let $\lambda : LH \stackrel{\cdot}{\Rightarrow} HL$ be a distributive law. We define $H^+ : Alg_L \to Alg_L$ as follows:

For any L-algebra (X, β_X), $H^+(X, \beta_X) = (HX, \beta_{HX})$, where $\beta_{HX} : L(HX) \to LX$ is defined by

$$\beta_{HX} \triangleq H\beta_X \circ \lambda_X$$

For any $f : (X, \beta_X) \to (Y, \beta_Y)$ L-algebra morphism, we define $H^+(f)$ as $H(f) : HX \to HY$. $H^+(f)$ is an algebra morphism from $(LX, H\beta_X \circ \lambda_X)$ to $(LY, H\beta_Y \circ \lambda_Y)$, since the following diagrams commute:

$$\begin{array}{ccccc} LHX & \xrightarrow{\lambda_X} & HLX & \xrightarrow{H\beta_X} & HX \\ {\scriptstyle LH(f)}\downarrow & (1) & {\scriptstyle HL(f)}\downarrow & (2) & \downarrow{\scriptstyle H(f)} \\ LHY & \xrightarrow[\lambda_Y]{} & HLY & \xrightarrow[H\beta_Y]{} & LY \end{array}$$

It is easy to check that H^+ is a functor on the category of L-algebras. Moreover, H^+ is a lifting of H along the forgetful functor $V : Alg_L \to \mathcal{C}$.

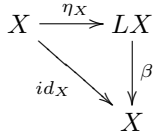
■

Using Theorem 3.3.6 above, one can easily show that a λ-bialgebra always induces a structured (co)algebra. In order to prove the converse, we need to consider the notions of pointed and copointed endofunctors, and the corresponding notions of (co)algebras. Following [52], we define

Definition 3.3.7 (Pointed Endofunctor) *A* pointed endofunctor $\langle L, \eta \rangle$ *on a category* $\mathcal{C}$ *is an endofunctor* L *on* $\mathcal{C}$ *together with a natural transformation* $\eta : Id \dot{\Rightarrow} L$.

An *algebra for a pointed functor* $\langle L, \eta \rangle$ is an L-algebra $\langle X, \beta \rangle$ such that the following *unit Law* for β hold:

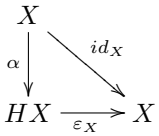
$$\beta \circ \eta_X = id_X$$



Definition 3.3.8 (Copointed Endofunctor) *A* copointed endofunctor $\langle H, \varepsilon \rangle$ *on a category* $\mathcal{C}$ *is an endofunctor* H *on* $\mathcal{C}$ *together with a natural transformation* $\varepsilon : H \dot{\Rightarrow} Id$.

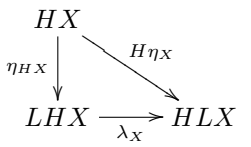
A *coalgebra for a copointed functor* $\langle H, \varepsilon \rangle$ is an H-coalgebra $\langle X, \alpha \rangle$ such that the following *counit Law* for α hold:

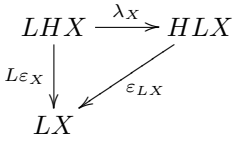
$$\varepsilon_X \circ \alpha = id_X$$



Definition 3.3.9 *A distributive law of a pointed endofunctor* $\langle L, \eta \rangle$ *over a copointed endofunctor* $\langle H, \varepsilon \rangle$ *is a natural transformation* $\lambda : LH \dot{\Rightarrow} HL$ *such that the following "coherence laws" hold:*

$$\lambda \circ \eta H = H\eta \qquad\qquad L\varepsilon = \varepsilon L \circ \lambda$$





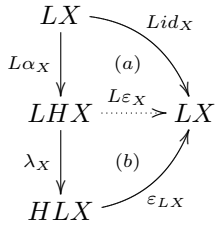
Theorem 3.3.10 *Let* $\langle L, \eta \rangle$ *be a pointed endofunctor on* $\mathcal{C}$*, and let* $\langle H, \varepsilon \rangle$ *be a copointed endofunctor on* $\mathcal{C}$*. Then the following notions are mutually equivalent.*

1. *Distributive laws* λ *of* L *over* H*.*
2. *Liftings* L^+ *of* L *to the* H*-coalgebras along the forgetful functor.*
3. *Liftings* H^+ *of* H *to the* L*-algebras along the forgetful functor.*

Proof. (1) $\Rightarrow$ (2) and (1) $\Rightarrow$ (3) Let $\lambda : LH \dot{\Rightarrow} HL$ be a distributive law of a pointed endofunctor $\langle L, \eta \rangle$ over a copointed endofunctor $\langle H, \varepsilon \rangle$. In addition to the proof of Theorem 3.3.6 we require to prove that the liftings are over categories of (co)algebras for *(co)pointed functors* and that the liftings are (co)pointed functors.

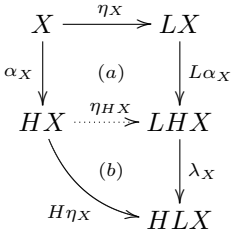
(1) $\Rightarrow$ (2) Let $L^+ : Coalg_H \to Coalg_H$ be the lifting of L defined in the proof of Theorem 3.3.6. L^+ turns out to be a functor over the category $Coalg_{\langle H,\varepsilon \rangle}$ of coalgebras for the *copointed*

endofunctors $\langle H, \varepsilon \rangle$. Let (X, α_X) be a coalgebra for $\langle H, \varepsilon \rangle$, we need to prove that $L^+(X, \alpha_X)$ satisfies the counit law:



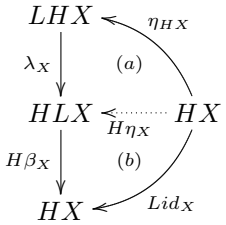
(a) commutes, by the counit law.
(b) commutes, since λ is a distributive law between a pointed and a copointed functors.

Now we prove that L^+ is itself a pointed endofunctor. Let (X, α_X) be an $\langle H, \varepsilon \rangle$-coalgebra, let $\eta^+_{(X,\alpha)} : (X, \alpha_X) \to L^+(X, \alpha_X)$ be defined as $\eta_X : X \to LX$. We left to show that η_X is a morphism between $\langle H, \varepsilon \rangle$-coalgebras.



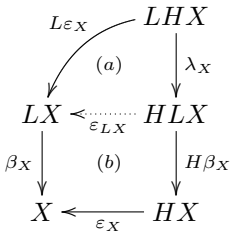
(a) commutes, from the natural transformation of η .
(b) commutes, by the coherence law for λ.

(1) $\Rightarrow$ (3) Dually, let $H^+ : Alg_L \to Alg_L$ be the lifting of H defined in the proof of Theorem 3.3.6. H^+ turns out to be a functor over the category $Alg_{\langle L,\eta \rangle}$ of algebras for the pointed endofunctors $\langle L, \eta \rangle$. Let (X, β_X) be a algebra for $\langle L, \eta \rangle$, we need to prove that $H^+(X, \beta_X)$ satisfies the unit law:



(a) commutes, since λ is a distributive law between a pointed and a copointed functors.
(b) commutes, by the unit law.

Now we prove that H^+ is itself a copointed endofunctor. Let (X, β_X) be an $\langle L, \eta \rangle$-algebra, let $\varepsilon^+_{(X,\beta)} : (X, \beta_X) \to L^+(X, \beta_X)$ be defined as $\varepsilon_X : HX \to X$. We left to show that ε_X is a morphism between $\langle L, \eta \rangle$-algebras.



(a) commutes, by the coherence law for λ.
(b) commutes, from the natural transformation of ε.

(2) $\Rightarrow$ (1) Conversely, consider a lifting L^+ of L to H-coalgebras, hence $U_H L^+ = LU_H$, where $U_H : Coalg_{\langle H,\varepsilon\rangle} \to \mathcal{C}$ is the forgetful functor. Let G_H be the left adjoint of $U_H, U_H \dashv G_H$. A distributive law λ of L over the endofunctor H can be defined as follows: first take the natural transformation $L\varepsilon : U_H L^+ G_H = LU_H G_H = LH \Rightarrow L$, then transpose it across the adjunction $U_H \dashv G_H$ obtaining $\bar{\lambda} : L^+ G_H \Rightarrow G_H L$, and finally define λ to be $U_H \bar{\lambda} : U_H L^+ G_H = LH \Rightarrow HL$. It is easy to prove that λ is a distributive law over H.

(3) $\Rightarrow$ (1) Dually, let H^+ be a Lifting of H, take $\eta_H : H \Rightarrow HL = HU^L G_L = U^L H^+ G^L$, transpose it across $G^L \dashv U^L$ obtaining $\bar{\lambda} : F^L H \Rightarrow L^+ G^L$, and finally define λ to be $U^L \bar{\lambda} : U^L G^L H = LH \Rightarrow HL = U^L H^+ G^L$.

■

3.3.1 Final Bialgebras

One of the main motivation for introducing λ-bialgebras is the fact that final coalgebras lift to final λ-bialgebras and initial algebras lift to initial λ-bialgebras. The proof of these facts is taken from [11]. The following lemmas are necessary to prove these facts:

Lemma 3.3.11 *Let $H : \mathcal{C} \to \mathcal{C}$ be a functor. Let 0 be an initial object of $\mathcal{C}$. Then there is a unique H-coalgebra structure on the initial object such that it yields an initial H-coalgebra.*

Proof. The unique initial morphism $!_{H0} : 0 \to H0$ gives a H-coalgebra structure on the initial object 0. For any H-coalgebra $\langle X, \alpha\rangle$ there is a unique initial morphism $!_X : 0 \to X$, and again by the initiality property, it is a coalgebra homomorphism.

$$\begin{array}{ccc} H0 & \xrightarrow{H!_X} & HX \\ {\scriptstyle !_{H0}}\uparrow & & \uparrow{\scriptstyle \alpha_X} \\ 0 & \xrightarrow[!_X]{} & X \end{array}$$

■

Dually, we have the following lemma:

Lemma 3.3.12 *Let $L : \mathcal{C} \to \mathcal{C}$. Let 1 be a final object of $\mathcal{C}$. Then there is a unique L-algebra structure on the final object such that it yields a final L-algebra.*

Proof. The unique final morphism $!_{L1} : L1 \to 1$ gives a L-algebra structure on a final object 1. For any L-algebra $\langle X, \beta\rangle$ there is a unique final morphism $!_X : X \to 1$, and again by the finality property, it is an algebra homomorphism.

$$\begin{array}{ccc} LX & \xrightarrow{L!_X} & L1 \\ {\scriptstyle \beta_X}\downarrow & & \downarrow{\scriptstyle !_{L1}} \\ X & \xrightarrow[!_X]{} & 1 \end{array}$$

■

Theorem 3.3.13 *Let $\lambda : LH \dot{\Rightarrow} HL$ be a distributive law of functor L over functor H. Then the initial L-algebra lifts to an initial λ-bialgebra.*

Proof. The proof follows from Lemma 3.3.11. ■

Theorem 3.3.14 *Let $\lambda : LH \dot{\Rightarrow} HL$ be a distributive law of functor L over functor H. Then the final H-coalgebra lifts to a final λ-bialgebra.*

Proof. The proof follows from Lemma 3.3.12. ∎

3.4 Congruences and Bisimulations

Initial morphisms, i.e. algebra morphisms from initial algebras, induce equivalences which are *congruences* w.r.t. algebra operations. Dually, *final morphisms*, i.e. coalgebra morphisms into final coalgebras, induce equivalences which have coinductive characterizations in terms of *bisimulations*.

Before introducing the notions of *H-bisimulation* and *L-congruence*, we introduce the notion of span:

Definition 3.4.1 *A* span $(\mathcal{R}, r_1, r_2)$ *on objects X, Y consists of an object $\mathcal{R}$ in $\mathcal{C}$, and two ordered arrows, $r_1 : \mathcal{R} \to X$ and $r_2 : \mathcal{R} \to Y$.*
Spans on objects X and Y can be ordered as follows:

$$(\mathcal{R}, r_1, r_2) \leq (\mathcal{R}', r_1', r_2') \iff$$

$$\exists f : \mathcal{R} \to \mathcal{R}'.\ \forall i = 1, 2.\ r_i = r_i' \circ f\ .$$

The notion of binary relation is expressed, in a general categorical setting, as an equivalence class of monic spans.

As pointed out in [78], H-bisimulations on H-coalgebras can be simply taken to be spans in the category of H-coalgebras. The following definition due to [78], generalizes the original definition of [6].

Definition 3.4.2 (H-bisimulation) *Let H be an endofunctor on the category $\mathcal{C}$. A span $(\mathcal{R}, r_1, r_2)$ on objects X, Y is an* H-bisimulation *on the H-coalgebras (X, α_X) and (Y, α_Y), if there exists an arrow of $\mathcal{C}$, $\gamma : \mathcal{R} \to H(\mathcal{R})$, such that $((\mathcal{R}, \gamma), r_1, r_2)$ is a coalgebra span, i.e.*

$$\begin{array}{ccccc} X & \xleftarrow{r_1} & \mathcal{R} & \xrightarrow{r_2} & Y \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \alpha_Y} \\ H(X) & \xleftarrow[H(r_1)]{} & H(\mathcal{R}) & \xrightarrow[H(r_2)]{} & H(Y) \end{array}$$

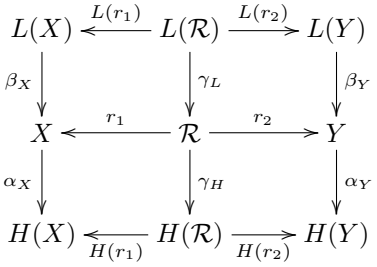
Definition 3.4.3 (L-congruence) *Let L be an endofunctor on the category $\mathcal{C}$. A span $(\mathcal{R}, r_1, r_2)$ on objects X, Y is an* L-congruence *on the L-algebras (X, β_X) and (Y, β_Y), if there exists an arrow of $\mathcal{C}$, $\gamma : L(\mathcal{R}) \to \mathcal{R}$, such that $((\mathcal{R}, \gamma), r_1, r_2)$ is an algebra span, i.e.*

$$\begin{array}{ccccc} L(X) & \xleftarrow{L(r_1)} & L(\mathcal{R}) & \xrightarrow{L(r_2)} & L(Y) \\ {\scriptstyle \beta_X}\downarrow & & \downarrow{\scriptstyle \gamma} & & \downarrow{\scriptstyle \beta_Y} \\ X & \xleftarrow[r_1]{} & \mathcal{R} & \xrightarrow[r_2]{} & Y \end{array}$$

In *Set*, one often only considers bisimulations which are relations, i.e. spans $(\mathcal{R}, \pi_1, \pi_2)$, where $\mathcal{R} \subseteq X \times Y$ is a relation. Notice that every span $(\mathcal{R}, r_1, r_2)$ in *Set* can be regarded as representing the relation amounting to the image $\langle r_1, r_2 \rangle(\mathcal{R}) \subseteq X \times Y$. The order on spans corresponds to relational inclusion of images. Furthermore, the image of a (span) bisimulation is a (relational) bisimulation (e.g. see [69], Lemma 5.3).

When the two H-coalgebras (X, α_X) and (Y, α_Y) in the definition above coincide, we will simply say that the span is an H-bisimulation on the H-coalgebra (X, α_X).

Definition 3.4.4 ($\langle L, H\rangle$-bicongruence) *Let L, H be endofunctors on the category $\mathcal{C}$. A span $(\mathcal{R}, r_1, r_2)$ on objects X, Y is an* $\langle L, H\rangle$–bicongruence *on the $\langle L, H\rangle$-bialgebras (X, β_X, α_X) and (Y, β_Y, α_Y), if there exist an algebraic structure $\gamma_L : L(\mathcal{R}) \to \mathcal{R}$, and a coalgebraic structure $\gamma_H : \mathcal{R} \to H(\mathcal{R})$, such that $((\mathcal{R}, \gamma_L, \gamma_H), r_1, r_2)$ is a bialgebra span, i.e.*



We call λ-bicongruence a bicongruence over λ-bialgebras.

Definition 3.4.5 (λ-bicongruence) *Let L, H be endofunctors on the category $\mathcal{C}$. A span $(\mathcal{R}, r_1, r_2)$ on objects X, Y is a* λ-bicongruence *on the λ-bialgebras (X, β_X, α_X) and (Y, β_Y, α_Y), if there exist an algebraic structure $\gamma_L : L(\mathcal{R}) \to \mathcal{R}$, and a coalgebraic structure $\gamma_H : \mathcal{R} \to H(\mathcal{R})$, such that $((\mathcal{R}, \gamma_L, \gamma_H), r_1, r_2)$ is a λ-bialgebra span.*

3.5 Bialgebraic Semantics

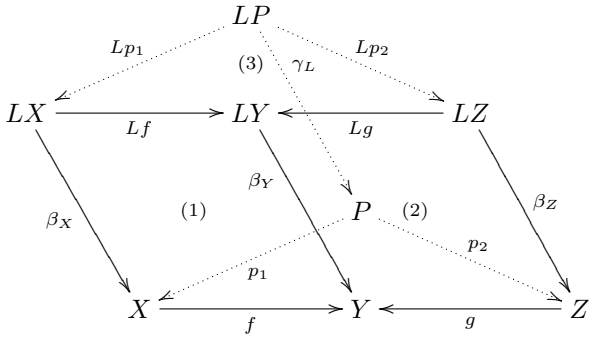
Initial algebraic semantics are morphisms from initial algebras; final coalgebraic semantics are morphisms into final coalgebras. As shown in [3], under suitable conditions, the equivalences induced by initial/final semantics are congruences/bisimulations. Here we recall basic definitions and results. We start by giving the following technical definition (for the definition of *weak pullback* see Appendix 7.3):

Definition 3.5.1 *The functor F preserves weak pullbacks if, for all weak pullbacks (P, p_1, p_2), $(F(P), F(p_1), F(p_2))$ is a weak pullback.*

3.5.1 Algebraic Semantics

Lemma 3.5.2 *Suppose that L preserves weak pullbacks. If $f : (X, \beta_X) \to (Y, \beta_Y)$ and $g : (Z, \beta_Z) \to (Y, \beta_Y)$ are morphisms, then the pullback of f and g in $\mathcal{C}$ is an L-congruence on (X, β_X) and (Z, β_Z)*

Proof. The proof follows immediately from Definition 3.5.1. There exists $\gamma_L : LP \to P$ because LP together with $\beta_X \circ Lp_1 : LP \to X$ and $\beta_Z \circ Lp_2 : LP \to Z$ is a cone for f and g.

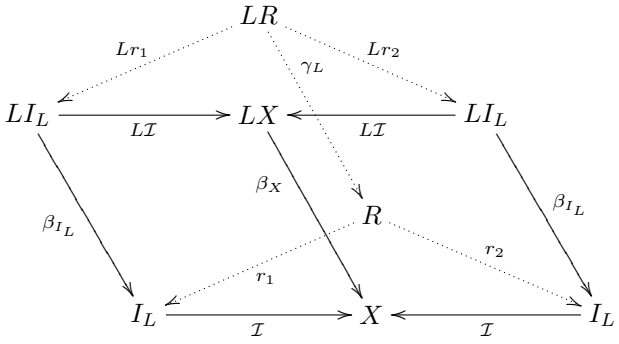


(1),(2) commutes, by algebraic homomorphisms.
(3) commutes, since there exists $\gamma_L : LP \to P$, such that (γ_L, p_1, p_2) is an algebraic span. ∎

The generalization in a categorical context of the set-theoretic notion of equivalence induced by a morphism on its domain is the notion of *kernel pair* (see Appendix 7.3) of a morphism. The following theorem ensures that *initial algebra* morphisms induce equivalences which are *congruences w.r.t.* the algebraic structure:

Theorem 3.5.3 *Suppose that $L : \mathcal{C} \to \mathcal{C}$ preserves weak pullbacks and it has an initial L-algebra (I_L, β_{I_L}). Let (X, β_X) be a L-algebra, and let $\mathcal{I} : (I_L, \beta_{I_L}) \to (X, \beta_X)$ be the unique initial morphism. Then, the kernel pair of $\mathcal{I}$ is a L-congruence.*

Proof. The proof follows from the Lemma 3.5.2 using definition of *kernel pair*. Let us denote (R, r_1, r_2) be the *kernel pair* of $\mathcal{I}$. There exists $\gamma_L : LR \to R$, such that following diagram commute:



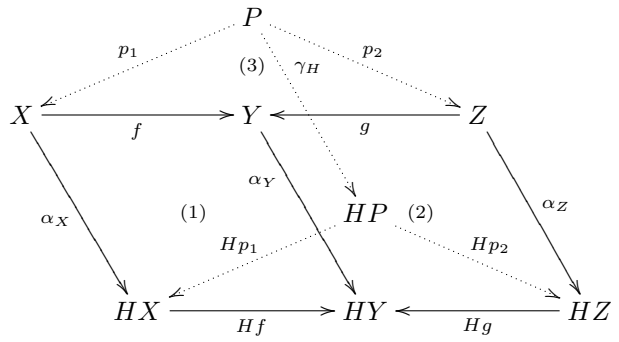
■

3.5.2 Coalgebraic Semantics

If the functor H preserves weak pullbacks, then pullbacks of H-coalgebra morphisms are H-bisimulations:

Lemma 3.5.4 *Suppose that H preserves weak pullbacks. If $f : (X, \alpha_X) \to (Y, \alpha_Y)$ and $g : (Z, \beta_Z) \to (Y, \alpha_Y)$ are morphisms, then the pullback of f and g in $\mathcal{C}$ is an H-bisimulation on (X, α_X) and (Z, α_Z)*

Proof. The proof follows immediately from Definition 3.5.1.



(1),(2) commutes, by coalgebraic homomorphisms.
(3) commutes, since there exists $\gamma_H : P \to HP$, such that (γ_H, p_1, p_2) is a coalgebraic span. ■

The following theorem generalizes the fact that, in set-theoretic categories, equivalences induced by unique morphisms into final coalgebras can be characterized coinductively as *greatest* H-bisimulations.

Theorem 3.5.5 *Suppose that $H : \mathcal{C} \to \mathcal{C}$ preserves weak pullbacks and it has a final H-coalgebra $(\Omega_H, \alpha_{\Omega_H})$. Let (X, α_X) be a H-coalgebra, and let $\mathcal{M} : (X, \alpha_X) \to (\Omega_H, \alpha_{\Omega_H})$ be the unique final morphism. Then*

(i). for all H-bisimulations $(\mathcal{R}, r_1, r_2)$ on (X, α_X), $\mathcal{M} \circ r_1 = \mathcal{M} \circ r_2$;

(ii). the kernel pair of $\mathcal{M}$ is an H-bisimulation on (X, α_X).

Proof. (i). The proof follows from Lemma 3.5.4 using the finality property.

(ii). The proof follows immediately from Lemma 3.5.4 and the definition of *kernel pair*.

■

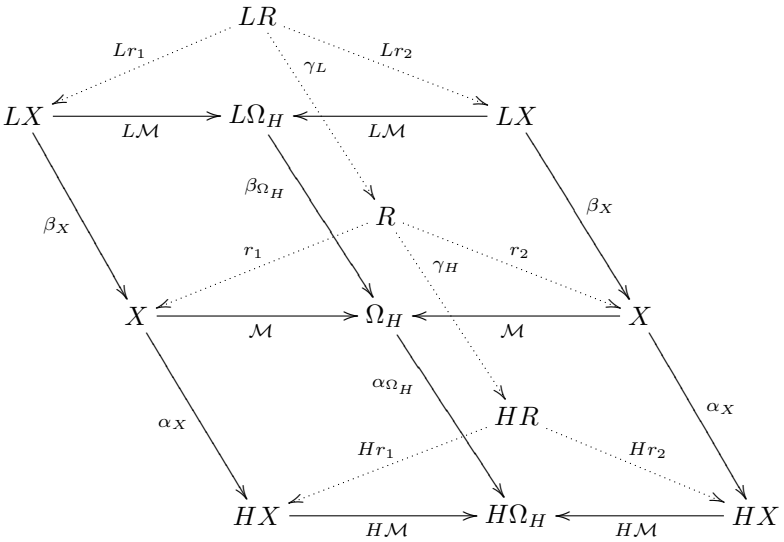
3.5.3 Bialgebraic Semantics

The main motivation for introducing λ-bialgebras (or structured coalgebras) is to extend in the bialgebraic setting Theorem 3.5.5 of Section 3.5.2 in such a way that the equivalence induced by the *final morphism* is both a *bisimulation* and a *congruence*. This follows from the following theorem proved by [11]:

Theorem 3.5.6 *Let λ be a distributive law of the functor L over the functor H. The greatest bisimulation $R \subseteq X \times Y$ between any two λ-bialgebras (X, β_X, α_X) and (Y, β_Y, α_Y) is a congruence.*

Lemma 3.5.7 *Let λ be a distributive law of the functor L over the functor H. Assume that there exists a final λ-bialgebra $(\Omega_H, \beta_{\Omega_H}, \alpha_{\Omega_H})$. Let (X, β_X, α_X) be a λ-bialgebra and let $\mathcal{M} : (X, \beta_X, \alpha_X) \to (\Omega_H, \beta_{\Omega_H}, \alpha_{\Omega_H})$ be the unique final morphism. If H preserves weak pullbacks, then the kernel pair of $\mathcal{M}$ is an $\langle L, H \rangle$-bicongruence on (X, β_X, α_X).*

Proof. The proof follows from Theorems 3.5.6, 3.5.5 and 3.3.14. There exists $\gamma_L : LR \to R$ and $\gamma_H : R \to HR$ such that $((\mathcal{R}, \gamma_L, \gamma_H), r_1, r_2)$ is a λ-bialgebra span.



■

4

Bialgebraic Specifications

The concept of algebraic specification originated in the pioneering work of Guttag, Liskov, Zilles, [36, 58]. For Object Oriented languages, Reichel [64] proposed to use *coalgebras* for specifying classes and objects. Recently [45], bialgebras have been proposed for specifying classes and objects of OO-Languages. In this chapter, we first present basic concepts of (co)algebraic specifications. Then, we focus on bialgebraic specifications of OO-Languages. We will introduce the notions of class specification and class implementation, together with various examples.

4.1 (Co)algebraic Specifications

An algebraic specification provides a description of both the syntax and semantics of an abstract data type, by stating its properties as axioms, which relate the operations of the data type to each other. Algebraic specification refers to the use of algebraic semantics and equational reasoning for functional systems [80, 81]. Following Ehrig, Mahr, "Fundamentals of Algebraic specifications 1", [31], we present here some basic notions about algebras and algebraic specifications.

Definition 4.1.1 *A* specification Γ *is a pair* (Σ, E)*, consisting of a signature* $\Sigma = (\Sigma_n)_{n\in\mathbb{N}}$*, i.e. a family of operation symbols (we write* $op : n$ *for* $op \in \Sigma_n$*) and a set* E *of* Σ*-equations.*

Definition 4.1.2 *A* Σ-algebra $A = \langle |A|, (op^A)_{op\in\Sigma}\rangle$ *consists of a carrier set* $|A|$ *and a family of operations such that* $op^A : |A|^n \to |A|$ *if* $op : n \in \Sigma$*.* Σ-algebras *can be endowed with the structure of a category,* Alg_Σ*, by defining a* Σ*-algebra homomorphism* $f : A \to B$ *(i.e.* $f : (|A|, (op^A)_{op\in\Sigma}) \to (|B|, (op^B)_{op\in\Sigma})$*) as a function* $f : |A| \to |B|$ *between carriers which respects the operations, i.e.* $op^B \circ f^n = f \circ op^A$*.*

$$\begin{array}{ccc} |A|^n & \xrightarrow{f^n} & |B|^n \\ {\scriptstyle op^A}\downarrow & & \downarrow{\scriptstyle op^B} \\ |A| & \xrightarrow[f]{} & |B| \end{array}$$

Let T_Σ be the term algebra over Σ and, for a given set X of variables, let $T_\Sigma(X)$ be the algebra of Σ-terms with variables in X. We denote the set of variables actually occurring in a term t by $\nu(t)$. The term algebra T_Σ is an initial object in Alg_Σ. An *assignment* for a set of variables X into a Σ-algebra A is a function $v : X \to |A|$. The term algebra $T_\Sigma(X)$ is free over X in Alg_Σ : if $v : X \to |A|$ is an assignment for X into A, its free extension is denoted by $\bar{v} : T_\Sigma(X) \to A$.

A Σ-equation (over X) is a pair of terms $s = t$ with $s, t \in T_\Sigma(X)$. For each assignment $v : X \to |A|$, if $\bar{v}(s) = \bar{v}(t)$, then Σ-equation is satisfied in a Σ-algebra A.

Definition 4.1.3 *An* algebraic specification $\Gamma = \langle \Sigma, E\rangle$ *is a specification for a given* Σ*-algebra* A*, if* A *satisfies all equations in* E*. Here* A *is called* Γ*-algebra.*

The idea behind algebraic specifications is that an abstract data type (ADT) can be described by giving

List(A) :
Signature :
$nil: \ 1 \to X$
$cons: \ A \times X \to X$
$head: \ X \to A$
$tail: \ X \to X$
$isEmpty: \ X \to \mathbb{B}$
Axioms :
$isEmpty(nil) = true$
$isEmpty(cons(a, l)) = false$
$head(cons(a, l)) = a$
$tail(cons(a, l)) = l$

Figure 4.1: Example of Specification for List

1. a signature, which is a list of the names and types of the ADT's operations (an operation's type is given by its domain and range), and

2. a set of equations (called axioms), which convey the intended meaning (i.e. semantics) of those operations. Usually, ADT's operations are modeled by (mathematical) functions.

$$\begin{cases} nil: \ 1 \to X \\ cons: \ A \times X \to X \end{cases} \qquad \begin{cases} head: \ X \to A \\ tail: \ X \to X \end{cases}$$

More generally, algebraic specifications can be defined not only for Σ-algebras, but for generic T-algebras, where T defines the type of the algebra operations. For instance, one can describe a list by giving their constructors *nil* and *cons* on the left above. An A-labelled list is either the empty list *nil* or a non-empty list (a, l) where l is a list and $a \in A$ is a label. Dually, we can define coalgebraic specifications, by considering coalgebraic signatures, where operations have type $X \to T$. A coalgebraic specification of A-labelled list is given on the right above. It gives the "destructors" *head* and *tail* operations, *head* evaluates the front of the list and *tail* evaluate the remaining list by removing the head from the given list. By considering a mixed algebraic/coalgebraic signature, we get a bialgebraic specification. An example of a bialgebraic specification for A-labelled lists is presented in Figure 4.1.

The algebraic part is given by the *constructors nil* and *cons*. The coalgebraic part consists of operations (*destructors*) *head*, *tail* and *isEmpty*. The behaviour of the operations is specified by the axioms in Figure 4.1.

As another example, let us consider the ADT stack. A stack is a "container" in which items are held one on top of another and in which both insertions *(push)* and deletions *(pop)* of items are done at the same end, called the top. Also, only the item occupying the top of the stack is observable.

$Stack(A)$: -the specification is parameterized by the data type of the elements to be stored in the stack

Signature :

new_1	$: \ 1 \to X$	-yields the empty stack
new_2	$: \ A \times X \to X$	-yields stack obtained by placing item at top
isEmpty	$: \ X \to \mathbb{B}$	-answers "Is the stack empty?"
top	$: \ X \to A$	-yields item at top of the stack
push	$: \ X \times A \to X$	-yields stack obtained by inserting item at top
pop	$: \ X \to X$	-yields stack obtained by deleting top item

In the above specification, new_1 and new_2 are constructors, while *isEmpty*, *top*, *push*, and *pop* are the destructors. An A-labelled stack is either the empty stack new_1 or build a non-empty stack

new_2 of the form (s, a) where s is the stack and $a \in A$ is a label. The destructor, *isEmpty* is used to know wheather the stack is empty or not, and *top* returns a a non-empty stack (s, a).

Now we apply these operations for stacks or as observing properties of stacks. Assume that A (say Z) is an integer, we can form expressions such as

1. $new_2(new_1, 6)$
   It creates a stack containing 6.

2. $push(push(pop(push(push(new_1, 3), 9)), 5), 2)$
   It denotes the stack containing 3, 5, and 2. (The 9 was pushed, but then popped.) Hence, a somewhat simpler expression intended to denote the same stack is
   $push(push(push(new_1, 3), 5), 2)$
   therefore one easily describe a stack containing $v_1, v_2, \ldots, v_n$, from bottom to top. Then the simplest expression for it would be
   $push(push(...(push(push(new_1, v_1), v_2), \ldots), v_{n-1}), v_n)$

3. $isEmpty(pop(push(pop(push(new_1, 5)), 2)))$
   It results true or false, according to whether the expression serving as the argument of *isEmpty* denotes the empty stack. In this case it is *true*.

Here are the axioms for Stack, which specify the behaviour of the operations.
Axioms:

1. $isEmpty(new_1) = true$
   It says that the stack has the property of being empty.

2. $new_2(s, x) = push(s, x)$
   It says that a stack can be obtained by inserting first element x to the stack s, making it not empty.

3. $isEmpty(push(s, x)) = false$
   It says that any stack obtained by pushing some element x onto some stack s is not empty.

4. $isEmpty(s) = true \Rightarrow top(s) = *$
   It says that the stack has the property of being empty means top of the stack is $*$.

5. $top(push(s, x)) = x$
   It says that the element x is on top of the stack obtained by pushing x onto any stack s.

6. $pop(push(s, x)) = s$
   It says that, if we pop a stack obtained by pushing some element x onto some stack s, we get the stack s as the result.

Does every possible stack is describable via an expression that is pop-free? The solution is *yes*. For instance, consider the following expression

$$pop(push(push(pop(push(push(new_1, 5), 2)), 0), 3))$$

It contains subexpressions of the form $pop(push(s, x))$. By Axiom (6), this subexpression is equivalent to s. Applying this at two places in the above expression, we get,

$$(push(push(new_1, 5), 0).$$

We can write the above "transformation" as follows:

$$pop(push(push(pop(push(push(new_1, 5), 2)), 0), 3))$$

$$= \langle\ Axiom(6),\ with\ s := push(new_1, 5)\ and\ x := 2\ \rangle$$

$$pop(push(push(push(new_1, 5), 0), 3))$$
$$= \langle\ Axiom(6),\ with\ s := push(push(new_1, 5), 0)\ and\ x := 3\ \rangle$$
$$push(push(new_1, 5), 0).$$

Hence, any expression satisfying the above axioms can be reduced to a *pop-free* expression.

We are given the more general case of bialgebraic specification. However, the interesting cases are those where the bialgebra is a λ-bialgebra.

4.2 Class Specifications and Class Implementations

A *class specification* is like an abstract class, in which signatures of constructors and methods declarations are given without their actual implementation, and also assertions are given x which put constraints on the behaviour of methods and constructors of the signature. Implementations of the constructors and the methods satisfying the assertions in the class specification, are given in the *class implementation*, also called concrete class or simply class. The essentials are put in the class specifications and the particulars are in the class implementation.

Formally, we define:

Definition 4.2.1 *A* **class specification** S *is a structure consisting of*

- *a finite set of* constructor declarations

  $$c : T_1 \times \ldots \times T_p \to X$$

  where T_i *is either a basic type or a symbol* X *denoting a class type;*

- *a finite set of* method declarations

  $$m : X \times T_1 \times \ldots \times T_q \to T_0$$

  where $T_i,\ i \neq 0$ *is either a basic type or* X*, while* T_0 *can be polynomial combination of* X *and basic type;*

- *a finite set of* assertions*, regulating the behaviour of the objects belonging to the class.*

The language for assertions is any first order language with constant symbols and function symbols for denoting constructors, methods and (extensions of) behavioural equivalences at all types. Typical assertions are equations, *e.g.* see [68] for more details.

A class (implementation) consists of attributes (fields), constructors and methods. Attributes and methods of a class can be either private or public. For simplicity, we assume all attributes to be private, and all methods to be public. We do not use a specific programming language to define classes, since we are working at a semantic level. Any programming language would do. In this perspective, a class will be represented by a set (of objects) X; a field f of type T is represented by a function $f : X \to T$; the code corresponding to a constructor declaration $c : \prod_{j=1}^{p} T_j \to X$ is given by a set-theoretic function $\beta : \prod_{j=1}^{p} T_j \to X$, while the code corresponding to a method declaration $m : X \times \prod_{j=1}^{q} T_j \to T_0$ is given by a set-theoretic function $\alpha : X \times \prod_{j}^{q} T_j \to T_0$.

Summarizing:

Definition 4.2.2 *A class* $C = \langle X, \{f_i : X \to T_i\}_{i=1}^{n}, \{c_i : \prod_{j=1}^{p_i} T_{ij} \to X\}_{i=1}^{h},\ \{m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}\}_{i=1}^{k} \rangle$ *is defined by*

- *a set of objects/states* X*;*
- *functions* $f_i : X \to T_i$ *representing fields;*
- *functions* $\beta_i : \prod_{j=1}^{p_i} T_{ij} \to X$ *implementing constructors* c_i*;*

- *functions* $\alpha_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}$ *implementing methods* m_i.

Definition 4.2.3 *A class C* implements *a specification S if constructor and method declarations correspond, and their implementations satisfy the assertions in S.*

Class specifications correspond to bialgebraic specifications, where the algebraic part is given by constructors, and the coalgebraic part is given by methods, when these are viewed in uncurried form, *i.e.* $m : X \to [\prod_{j=1}^{q} T_j \to T_0]$. In Section 5.2 of Chapter 5, we will extensively discuss the categorical bialgebraic account of class specifications and classes.

4.3 Examples of Class Specifications

In this section we present four examples of bialgebraic specifications, see Table 4.1. We start with a specification of a class *Stack(A)*, which is borrowed from [49], and specifies the recursive data type of stacks with elements in A. As it is customary in OO-languages such as `Java`, we use the same name *new* for all constructors, which differ in the number and type of their parameters. The symbol $\approx$ in the assertions denotes equality on objects of class type. All methods in this example are unary. Notice the use of the "dot-notation" for method calls.

Our second example of class specification, *Register*, features binary method *eq* for comparing the content of two registers, and unary methods *set* and *get*. The method *set* sets the value of a register, giving a new state; using the method *get* we can get the value of a register. In the algebraic part, we have constructor *new* forming an element of type *Register*. One should read the equation $\mathrm{r.set}(n).\mathrm{get} = n$ as: if one sets state r the set message 'set' with parameter n and then asks for the value 'get', then the outcome is same as the value n. And the equation $\mathrm{r_1.get} = \mathrm{r_2.get} \Leftrightarrow \mathrm{r_1.eq(r_2)} = \mathit{true}$ as: if one asks the value 'get' of the state r_1 is 'equal to' the value 'get' of the state r_2, then the outcome is same as asking the state r_1 for equality 'eq' with parameter r_2 is 'true'. The equation $\mathrm{new.get} = 0$ of the specification mentions that the value of a newly created object of *Register* class is 0.

The class specification *λ-calculus* in Table 4.1, which generalizes the one in [41], represents the recursive data type of λ-terms under lazy evaluation, see [2]. There are three constructors, corresponding to the syntax of λ-terms: variable, application and abstraction, namely

$$M ::= z \mid MM \mid \lambda z.M$$

for z ranging over an infinite set of variables. Method *isval* tests whether a λ-term converges, i.e. it reduces to an abstraction, according to the leftmost strategy. Lazy convergence, denoted by $\Downarrow$, is defined as follows:

$$\frac{}{\lambda z.M \Downarrow \lambda z.M} \qquad \frac{M[N_1/z]N_2 \ldots N_k \Downarrow P}{(\lambda z.M)N_1 \ldots N_k \Downarrow P}$$

The first assertion in the class specification of λ-calculus expresses the fact that method *app* is a binary method which behaves like the constructor for application. The second assertion is used to axiomatize the notion of convergence.

As we will see, the class specification concerning λ-calculus is intended to give the standard notion of *lazy* observational equivalence when restricted to *closed* λ-terms.

A more sophisticated example of a generalized binary method is given by the cell-component of a cellular automaton. In the general case, where neighborhoods can vary at each generation, they can be best specified using sets of cells.

In Table 4.2, we present the classes T, R, Λ, C, implementing the corresponding class specifications. Constructor codes are omitted. Notice that the code in the implementation of Λ is not effective, since it uses the predicate $\Downarrow$ as a primitive, which is only semidecidable. It couldn't be otherwise.

Figure 4.2 shows an implementation of the class *Stack* in language *Fickle*, and following the syntax of *OCaml* [53], Figure 4.3 shows an implementation of the class *Register*.

class spec : *Stack(A)*
  constructors :
    new : $1 \to X$
    new : $X \times A \to X$
  methods :
    push : $X \times A \to X$
    pop : $X \to X$
    top : $X \to 1 + A$
  assertions :
    s.push(a).top $= a$
    s.push(a).pop $\approx$ s
    s.top $= * \Rightarrow$ s.pop $\approx$ s
    new.top $= *$
    new(s, a) $\approx$ s.push(a)
end class spec

class spec : *Register*
  constructors :
    new : $1 \to X$
  methods :
    set : $X \times N \to X$
    get : $X \to N$
    eq : $X \times X \to \mathbb{B}$
  assertions :
    r.set(n).get $= n$
    r_1.get $= r_2$.get $\Leftrightarrow$
      r_1.eq(r_2) $= true$
    new.get $= 0$
end class spec

class spec : λ-*calculus*
  constructors :
    new : $X \times X \to X$
    new : $Var \to X$
    new : $Var \times X \to X$
  methods :
    isval : $X \to \mathbb{B}$
    app : $X \times X \to X$
  assertions :
    new(M, N) $\approx M$.app(N)
    M.isval $= true \Leftrightarrow$
      $\exists z N.\ M \approx new(z, N)$
end class spec

class spec : *Cell*
  constructors :
    new : $N \times N \times State \to X$
  methods :
    $\text{get}_x : X \to N$
    $\text{get}_y : X \to N$
    $\text{get}_{state} : X \times State \to X$
    $\text{set}_{\text{neighborhood}} : X \times \mathcal{P}(X) \to X$
    $\text{set}_{\text{neighborhood}} : X \to \mathcal{P}(X)$
  assertions :
    ...
end class spec

Table 4.1: Examples of Class Specifications.

class T
  attributes :
    $first : 1 + A$
    $next : T$
  constructors :
    ....
  methods :
    s.push(a) = s$'$
      where s$'$.$first = a$ and
        s$'$.$next$ = s
    s.pop = **if** s.$first = *$
      then s **else** s.$next$
    s.top = s.first
end class

class R
  attributes :
    $val : int$
  constructors :
    ....
  methods :
    r.get = r.val
    r.set(n) = r$'$
      where r$'$.$val = n$
    r_1.eq(r_2) = **if** (r_1.get = r_2.get)
      then $true$
      else $false$
end class

class Λ
  attributes :
    $term : \lambda$-$string$
  constructors :
    ....
  methods :
    M.isval = **if** $M.term \Downarrow$
      then $\langle true, M\rangle$
      else $\langle false, M\rangle$
    M.app(N) = P
      where $P.term = (M.term)(N.term)$
end class

class C
  attributes :
    $val_x : int$
    $val_y : int$
    neighborhood : $\mathcal{P}(X)$
    state : $State$
  constructors :
    ....
  methods :
    ....
end class

Table 4.2: Examples of Classes Implementing the Class Specifications.

```
class StackException extends Exception{
}
abstract root class Stack extends Objects{
      abstract void push(int i){Stack};
      abstract void pop(){Stack};
      abstract int top(){Stack};
      abstract bool isEmpty(){Stack} }
state class EmptyStack extends Stack{
      void push(int i){Stack}{
         this⇓NonEmptyStack; element:= i;
         next := new EmptyStack; }
      void pop(){}{ throw new StackException; }
      int top(){}{ throw new StackException; }
      bool isEmpty(){} { return true; } }
state class NonEmptyStack extends Stack{
      int element;
      Stack next;
      void push(int i){}{
         NonEmptyStack second:= new NonEmptyStack;
         element:= i; next:=second; }
      void pop(){Stack}{
         int value := element;
         if (this.next.isEmpty()) {
           this⇓EmptyStack; }
         else {
           this.element := ((NonEmptyStackthis.next).element;
           this.next := ((NonEmptyStackthis.next).next; }  }
      int top(){Stack}{
         int value := element; return value }
      bool isEmpty(){} { return false; } }
```

Figure 4.2: Implementation of class *Stack* in *Fickle*

```
# class reg1 =
    object (self: 'a)
      val mutable rp = 0
      method getx = rp
      method setx x = rp ← x
      method eq (p: 'a) = rp = p#getx
    end;;
```

Figure 4.3: Implementation of class *Register* in *OCaml*

III

Third Part: Binary Methods

5

Bialgebraic Semantics for Binary Methods

In the coalgebraic approach of [64, 47, 49], a class is modeled as an F-coalgebra $(A, f : A \to F(A))$ for a suitable functor F. The carrier A represents the space of objects, and the coalgebra operation f represents the *public* methods of the class, *i.e.* the methods which are accessible from outside the class. Methods are viewed as functions acting on objects. The coalgebraic model, *i.e.* the unique morphism into the *final F-coalgebra*, induces precisely the *behavioural equivalence* on objects, whereby two objects are equated if, for each public method, the application of the method to the two objects, for any list of parameters, produces equivalent results. A benefit of this coalgebraic approach is a *coinduction principle* for establishing behavioural equivalence.

In this chapter, in order to account also for class *constructors*, we introduce an algebra part in our model, thus modelling classes as *bialgebras*. A similar move appears in [68, 23].

Binary methods, *i.e.* methods with more than one class parameter, apparently escape Reichel-Jacobs co(bi)algebraic approach. Namely, the extra class parameters produce contravariant occurrences in the functor modelling class methods, and hence cannot be dealt with by a straightforward application of the standard coalgebraic methodology.

In this chapter, we present a bialgebraic model of class specifications and implementations as defined in Chapter 4, Section 4.2. First, we discuss the case of unary methods, then we extend Reichel-Jacobs approach to *generalized binary methods*, these are methods whose type parameters are built over constants and class variables, using products, sums and the powerset type constructor. This is a quite large collection of methods, including all the methods which are commonly used in Object Oriented Programming.

Our focus of interest are equivalences on objects which are "well-behaved", *i.e.* are *congruences w.r.t.* method application. Hence they induce a minimal implementation of the given class specification, by considering the quotient of the class through the equivalence.

In this chapter, we show that canonical models can be built also for classes with generalized binary methods using purely covariant tools, at least in the case of *finitary binary methods*, *i.e.* methods where type constructors range over *finite* product, sum, and powerset. We propose two different solutions. Our first solution applies to the case where we already have a class implementation. It is based on the observation that the behaviour of a generalized binary method can be captured by a bunch of unary methods obtained after a suitable manipulation of the original method. The key step is that of *"freezing"*, in turn, the types of the class parameters to the states of the class implementation given at the outset, *i.e.* by viewing them as constant types.

Our second solution is based on a set-theoretic understanding of functions, whereby binary methods in a class specification can be viewed as *graphs* instead of functions. Thus contravariant function spaces in the functor are rendered as covariant sets of relations.

We prove that the bisimilarity equivalence induced by the *"freezing approach"* amounts to the *greatest congruence w.r.t.* method application on the given class, at least for finitary binary methods. As a by-product, we provide a (coalgebraic) coinduction principle for reasoning about such greatest congruence.

As far as the graph model is concerned, the bisimilarity equivalence is not a congruence, in general, even for finitary binary methods. The graph approach, however, yields an equivalence which always includes the freezing equivalence. Therefore, but somewhat remarkably, a necessary and sufficient condition for the graph bisimilarity to be a congruence is that the graph and freezing equivalence coincide. As a consequence, when this is the case, we obtain a spectrum of coinduction principles for reasoning on the greatest congruence.

We present various non-trivial examples of class specifications and implementations, where the graph bisimulation is a congruence.

A natural question to ask when the freezing approach does not coincide with the graph approach is why is it the case. We do not have a fully satisfactory answer, but we feel that this is a telltale alarm. Something is underspecified in the public interface of the class. A similar comment can be made when no maximal congruence exists in the realm of infinitary binary methods.

We emphasize the fact that, in the case of finitary binary methods, we do provide satisfactory canonical models, which can be conveniently understood in terms of final coalgebras, for suitable derived functors. The existence of a final coalgebra is important, since it provides a canonical implementation of a given specification. Since we do not want to introduce unnecessary restrictions due to the choice of our ambient category, we work in the category of sets and proper classes [6, 32], where all endofunctors can be shown to have a final coalgebra, see Chapter 3, Section 3.2.1. Thus, throughout the chapter, we fix $\mathcal{C}$ to be a category whose objects are the sets and classes of a (wellfounded or non-wellfounded) set-theoretic universe, and whose morphisms are the functions between them.

Comparison with Related Work

There are other approaches in the literature which address the problem of extending the coalgebraic model to binary methods. We offer the following analysis.

In [48], binary methods are allowed only when they are definable in terms of the unary methods of the class. Hence, in particular binary methods do not contribute to the definition of the observational equivalence. The same observation applies to the approach of [38], where binary methods are defined as algebraic extensions, thus only the case where the resulting type is the class itself is considered. Our approach is more general, since we do not require any connection *a priori* between binary and unary methods in the class.

Binary methods in full generality have been extensively studied in [76, 77], where various classes of mixed covariant-contravariant functors have been considered, and a theory of coalgebras and bisimulations has been studied for such functors. Tews' approach is very interesting, but quite different from our approach, since we use only covariant tools, from the very outset. Nonetheless, there are interesting connections between the two approaches. We consider also the powerset type constructor, which Tews does not include, but, apart from this, our generalized binary methods should correspond, essentially, to the class of *extended polynomial functors* of Tews[1]. Similarly, our finitary generalized methods should correspond to Tews' *extended cartesian functors*. Tews' bisimulations amount to congruence relations, and do not give rise, in general, to a coinduction principle, since the union of all congruences fails to be a congruence. However, for extended cartesian functors, the union of all congruences is again a congruence, [63, 77]. Our notion of freezing bisimulation is weaker, in the sense that any bisimulation in the sense of Tews is a freezing bisimulation, but not vice versa. Moreover, our notion of bisimulation, being monotone, gives always rise to a coinduction principle. However, the greatest freezing bisimulation fails, in general, to be a congruence. It is a congruence (the greatest one, in fact) exactly in the case of finitary methods. Thus, in this case, our notion of bisimilarity equivalence coincides with the one by Tews.To conclude this comparison, we make the following two remarks. First, our approach is more elementary. By modeling binary methods with purely covariant functors, we can reuse the standard coalgebraic machinery. On the other hand, Tews develops a theory of coalgebras and

[1] Apparently, Tews' extended polynomial functors cover a wider collection of methods, but we conjecture that also those particular cases which appear to escape from our approach should be recovered using manipulations similar to those introduced in Section 5.4. However, more work needs to be done.

bisimulations for mixed functors, which has interest also in itself. Second, in our setting, final coalgebras always exist, and hence we have canonical models, while in [77] mixed functors do not admit final coalgebras.

Yet another approach in the theory of coalgebraic semantics consists in avoiding binary methods altogether by considering a whole system of objects in place of single objects, *e.g.* by considering a class representing a list of points, in place of a class for a single point, see [13, 46]. This approach is quite different from ours.

Finally, there is an interesting connection between our approach and the approach of *hidden algebras*, see [34, 67], where the focus is on behavioural congruences, rather than on bisimulations. Our freezing model has the positive features of both approaches: the behavioural equivalence that we define is *both* a greatest bisimulation and the greatest congruence *w.r.t.* method application.

5.1 Generalized Binary Methods

In this chapter, we will focus on class implementations and class specifications as defined in Chapter 4, Section 4.2, and we assume that the methods appearing in such classes are *generalized binary*. Generalized binary methods are methods, whose parameter types are *(infinitary) generalized*. According to the following definition, *finitary generalized types* correspond to polynomial types extended with finite powerset, while *(infinitary) generalized types* extend the previous class of types with possibly infinitary sums, products and powerset constructors.

Definition 5.1.1 (Generalized Parameter Types)

- Finitary Generalized Types *range over the following grammar:*

$$(\mathcal{T}_{\mathcal{F}} \ni)\ T ::= \ X \mid K \mid T \times T \mid T + T \mid \mathcal{P}_f(T)\ ,$$

  where $X \in TVar$ is a variable for class types, and K is any constant type. Constant types include Unit, denoted by 1, Boolean, denoted by $\mathbb{B}$, Integer, denoted by $\mathbb{N}$.

- (Infinitary) Generalized Types *range over the following grammar:*

$$(\mathcal{T} \ni)\ T ::= \ X \mid K \mid \prod_{i \in I} T_i \mid \sum_{i \in I} T_i \mid \mathcal{P}(T)\ ,$$

  where I is a possibly infinite set of indices.

Notice that the product type $\prod_{i \in I} T_i$ in Definition 5.1.1 above subsumes the function space $K \to T$. That is, we allow functional parameters, where variable types can appear only in *strictly positive* positions.

For simplicity, in this chapter we will consider only one class in isolation. There would be no additional conceptual difficulty in dealing with the general case.

Throughout the chapter, we fix the following terminology:

Definition 5.1.2 (Binary Methods) *Let $m : X \times T_1 \times \ldots \times T_q \to T_0$ be a method, with $T_0 \in \mathcal{T}$. Then*

- *m is* (generalized) binary *if $T_1, \ldots, T_q \in \mathcal{T}$;*
- *m is* finitary binary *if $T_1, \ldots, T_q \in \mathcal{T}_{\mathcal{F}}$;*
- *m is* simple binary *if $T_1, \ldots, T_q$ are either constants or the class type X;*
- *m is* unary *if $T_1, \ldots, T_q$ are all constant types.*

Notice that, our simple binary methods correspond to ordinary binary methods. Throughout the chapter, *generalized binary methods* will be often simply called *binary methods*.

5.2 Bialgebraic Description of Objects and Classes: unary case

In this section, we illustrate the bialgebraic description of class specifications and class implementations of Chapter 4, Section 4.2, in the case of unary methods. We extend the coalgebraic description of [64, 47] with an algebra part modelling class constructors. A similar move appears also in [68, 23].

We start by explaining how a class specification induces a pair of functors.

Each constructor declaration $c : \prod_{j=1}^{p} T_j \to X$ in a class specification determines a functor $L : \mathcal{C} \to \mathcal{C}$ defined by

$$LX = \prod_{j=1}^{p} T_j. \tag{5.2.1}$$

In this way, $c : LX \to X$ will induce an L-algebra structure on X.

The treatment of methods is more indirect. By currying the type in a method declaration $m : X \times \prod_{j=1}^{q} T_j \to T_0$, we get the type $X \to [\prod_{j=1}^{q} T_j \to T_0]$. Thus, we define the functor $H : \mathcal{C} \to \mathcal{C}$ induced by m as follows:

$$HX \triangleq \prod_{j=1}^{q} T_j \to T_0. \tag{5.2.2}$$

Thus m will induce a H-coalgebra structure on X.

Notice that the functor H is a welldefined covariant functor, only if the method m is unary. Binary methods, such as the method *eq* in the class specification *Register*, or *app* in *λ-calculus* (see Chapter 4, Table 4.1), produce contravariant occurrences of X in the corresponding functor. For example, the functor induced by *eq* would be $H_{eq}X \triangleq X \to \mathbb{B}$. The coalgebraic approach does not apply directly to the case of binary methods. In Section 5.3, we discuss how to overcome this problem. Here we focus on the unary case. In this case, we can immediately associate a pair of functors to a class specification as follows:

Definition 5.2.1 *Let S be a class specification with constructor declarations $c_i : \prod_{j=1}^{p_i} T_{ij} \to X$, $i = 1, \ldots, h$ and with method declarations $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}$, $i = 1, \ldots, k$, where all methods are unary. The constructor declarations in S induce the functor $L : \mathcal{C} \to \mathcal{C}$ defined by*

$$L \triangleq \prod_{i=1}^{h} L_i \ ,$$

where $L_i : \mathcal{C} \to \mathcal{C}$ is the functor determined by the constructor declaration c_i defined as in (5.2.1). The method declarations in S induce the functor $H : \mathcal{C} \to \mathcal{C}$ defined by

$$H \triangleq \prod_{i=1}^{k} H_i \ ,$$

where $H_i : \mathcal{C} \to \mathcal{C}$ is the functor determined by the method declaration m_i, defined as in (5.2.2).

A class implementation induces a bialgebra for the functors determined by its constructor and method declarations, as follows:

Definition 5.2.2 *A* class $C = \langle X, \{f_i : X \to T_i\}_{i=1}^{n}, \{c_i : \prod_{j=1}^{p_i} T_{ij} \to X\}_{i=1}^{h}, \{m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0}\}_{i=1}^{k} \rangle$ *induces a bialgebra (X, β, α) (where α and β are defined below) for the functor pair $\langle L, H \rangle$ determined by the declarations of constructors and methods as in Definition 5.2.1 above:*

- *the algebra map $\beta : LX \to X$ is defined by $\beta \triangleq [\beta_i]_{i=1}^{h}$, where $\beta_i : L_iX \to X$ is the function implementing the constructor c_i, and $[\ ]$ denotes the standard case function;*

- *the coalgebra map* $\alpha : X \to HX$ *is defined by* $\alpha \triangleq \langle \alpha_i \rangle_{i=1}^{k}$, *where* $\alpha_i : X \to H_i X$ *is the function implementing the method* m_i, *and* $\langle\ \rangle$ *denotes the standard pairing functor.*

Thus, class implementations corresponding to a given specification can be viewed as bialgebras as follows:

Definition 5.2.3 *Let* S *be a class specification inducing a functor pair* $\langle L, H \rangle$. *A* class *implementing* S *is an* $\langle L, H \rangle$*-bialgebra satisfying the assertions in* S.

Notice that, in Definition 5.2.3 above, classes are taken up to fields, because these are private.

5.2.1 Coalgebraic Behavioural Equivalence

In this section, we characterize the behavioural equivalence on objects induced by the coalgebraic part of a class implementation.

A preliminary step in discussing behavioural equivalences and congruences consists in extending the behavioural equivalence on the set of objects X of a class to the whole structure of (sets interpreting) types over X. Such extension is defined through the following definition, which extends the notion of relational lifting of [40] to the powerset. In the definition below, by abuse of notation, we do not distinguish between types and their usual set-theoretic interpretation.

Definition 5.2.4 (Relational Lifting) *Let* R^X *be a relation on* X, *let* $T \in \mathcal{T}$ *be such that* $Var(T) \subseteq \{X\}$. *We define the extension* $R^T \subseteq T \times T$ *by induction on* T *as follows:*

- *if* $T = K$, *then*
$$R^T = Id_{K \times K},$$
- *if* $T = \prod_{i \in I} T_i$, *then*
$$R^T = \{(\vec{a}, \vec{a}') \mid \forall i \in I. a_i R^{T_i} a_i'\},$$
- *if* $T = \sum_{i \in I} T_i$, *then*
$$R^T = \{((i, a), (i, a')) \mid i \in I \wedge a R^{T_i} a'\},$$
- *if* $T = \mathcal{P}(T_1)$, *then*
$$R^T = \{(u, u') \mid \forall a \in u\ \exists a' \in u'.\ a R^T a'\ \wedge\ \forall a' \in u'\ \exists a \in u.\ a R^T a'\}.$$

In what follows, by abuse of notation, we will often denote the lifted relation R^T simply by R, when its type is clear from the context.

A strong motivation for the coalgebraic account of objects is that the quotient by the bisimilarity equivalence of a given class, when viewed as a coalgebra, can yield, in many cases, such as that of unary methods, a new model of the same class. For this to hold, we need at least that the bisimilarity equivalence is a *congruence w.r.t. method application, i.e.:*

Definition 5.2.5 (Congruence) *Let* $\approx^X$ *be an* equivalence *on the set of objects* X *of a class* C, *and let* $m : X \times T_1 \times \ldots \times T_q \to T_0$ *be a method in* C *implemented by* α, *then* $\approx^X$ *is a* congruence *w.r.t.* m *if*

$$x \approx^X x'\ \wedge\ a_1 \approx^{T_1} a_1' \wedge \ldots \wedge a_q \approx^{T_q} a_q'\ \Longrightarrow\ \alpha(x)(\vec{a}) \approx^{T_0} \alpha(x')(\vec{a}')\ ,$$

where $\approx^{T_i}$ *denotes the extension of* $\approx^X$ *to the type* T_i, *according to the definition above.*

Finally, having defined the coalgebraic account of a class the way we did, we have that the coalgebraic equivalence in the unary case equates objects with the same behaviour under application of methods:

Proposition 5.2.6 (Coalgebraic Bisimilarity Equivalence) *Let S be a class specification and let $(X, [\beta_i]_{i=1}^h, \langle\alpha_i\rangle_{i=1}^k)$ be an $\langle L, H\rangle$-bialgebra implementing S. Then*

(i). An H-bisimulation on $(X, \langle\alpha_i\rangle_i)$ is a relation $R \subseteq X \times X$ satisfying

$$x\,R\,x' \Longrightarrow \forall\alpha_i.\ \forall\vec{a}.\ \alpha_i(x)(\vec{a})\,R\,\alpha_i(x')(\vec{a})\ .$$

(ii). The coalgebraic bisimilarity equivalence $\approx_H$*, i.e. the greatest H-bisimulation on $(X, \langle\alpha_i\rangle_i)$, can be characterized as follows:*

$$x \approx_H x' \iff \forall\alpha_i.\ \forall\vec{a}.\ \alpha_i(x)(\vec{a}) \approx_H \alpha_i(x')(\vec{a})\ .$$

In particular, the following coinduction principle *holds:*

$$\frac{R \text{ is an } H\text{-bisimulation on } (X, \langle\alpha_i\rangle_i) \qquad x\,R\,x'}{x \approx_H x'}$$

Proof. By definition of coalgebraic bisimulation (see Definition 3.4.2 of Chapter 3). ∎

Thus we have also:

Theorem 5.2.7 *$\approx_H$ is the* greatest congruence *w.r.t. methods.*

Proof. Since all methods are unary, by definition of relational lifting on constant types, we immediately have that $\approx_H$ is a congruence *w.r.t.* methods. The fact that $\approx_H$ is the greatest congruence follows by observing that any congruence *w.r.t.* methods is an H-bisimulation. ∎

As we remarked earlier, a strong point of the coalgebraic approach to classes is that bisimilarity equivalences naturally yield, via quotienting, classes of the same signature as the original class, and furthermore preserve various kinds of assertions. This can be expressed also by saying that a suitable subcoalgebra of the final coalgebra still provides an implementation of the specification, in fact the canonical one.

It goes without saying that in dealing with bialgebras we would like to preserve the above important feature of the purely coalgebraic approach. To this aim, we need that final bialgebras exist, and furthermore that the behavioural equivalence is a congruence also *w.r.t.* constructors. In [78, 27], general conditions on categories of bialgebras are studied in order to ensure the above properties (see Chapter 3, Section 3.5.3).

These results can be extended/adapted also to the collection of functors modelling generalized binary methods considered later in this chapter, at least for assertions of a simple equational shape. In this case, if for a given bialgebra satisfying the assertions there is a "tight connection" between the algebraic and the coalgebraic structure, then the corresponding functors admit final bialgebras still satisfying the assertions, and the behavioural equivalence is a congruence *w.r.t.* constructors. Here we do not elaborate more on this issue, but we rather focus on the coalgebraic part, which is the most problematic one.

5.3 Coalgebraic Description of Generalized Binary Methods

In this section, we show how to extend the bialgebraic approach to binary methods. Our first proposal (Section 5.3.1) applies when a concrete bialgebra (*i.e.* class implementation) is already available. It is based on the observation that the behaviour of a binary method can be simulated by a bunch of unary methods, each one determined by "freezing" all the occurrences of X in the parameter types and object type, but one. The bunch being obtained after suitable manipulations of the original method. "Freezing" an occurrence of X means that X is replaced by the carrier, *i.e.* the set of states, of the given class. This allows us to define a covariant *freezing* functor F, where the contravariant occurrences in the original generalized binary method are replaced by a constant

type, namely the carrier of the given bialgebra. The freezing procedure is carried out in such a way that, at least in the case of finitary binary methods, the bisimilarity equivalence induced by F turns out to be the greatest congruence *w.r.t.* the original binary methods.

In Section 5.3.2, we present an alternative solution to the freezing functor, which we call *graph functor*. Here we turn contravariant occurrences in the type of parameters of a generalized binary method m into covariant ones simply by interpreting m as a *graph* instead of a function. To this aim, we introduce a new functor G (*graph functor*), where the function space is substituted by the corresponding space of *graph relations*.

The advantage of this latter solution with respect to the previous one is that this approach directly applies to specifications. The drawback is that the graph bisimilarity equivalence is not a congruence *w.r.t.* method application in general. One may wonder as to why this is the case. There is as yet no general explanation. Often this means that the specification is *under-determined*, or alternatively, there exist class implementations without a common refinement. However, there are many interesting situations where the graph equivalence is a congruence *w.r.t.* methods. In these cases a rich spectrum of conceptually independent coinduction principles is available. We discuss this issue in Section 5.3.3, together with the comparison of freezing and graph bisimilarity equivalences.

Throughout this section, let S be a class specification with constructors $c_i : \prod_{j=1}^{p_i} T_{ij} \to X,\ i = 1, \ldots, h$, and methods $m_i : X \times \prod_{j=1}^{q_i} T_{ij} \to T_{i0},\ i = 1, \ldots, k$. Moreover, let L be the functor induced by the constructors.

5.3.1 The Freezing Functor

Given a class implementation C, with carrier $\bar{X}$, we transform C into a class C^* containing only unary methods. To this aim, we proceed in two steps.

First, we reduce each binary method m to a bunch of *simple* binary methods with the same observable behaviour of m. To this aim, we proceed by processing parameters of complex types as follows. For each parameter of type $\sum_{i \in I} T_i$ in m, we consider methods $\{m_i\}_{i \in I}$, where the method m_i has a parameter of type T_i. Each parameter of type $\prod_{i \in I} T_i$ can be viewed as the product of $|I|$ parameters. More subtle is the treatment of parameters of type $\mathcal{P}(T')$. If m has a parameter of type $\mathcal{P}(T')$, *i.e.* $m : X \times \ldots \times \mathcal{P}(T') \times \ldots \to T_0$, then the behaviour of m can be simulated by a pair of methods $m_1 : X \times \ldots \to T_0$, where the parameter of type $\mathcal{P}(T')$ disappears, and $m_2 : X \times \ldots \times \mathcal{P}(T'[\bar{X}|X]) \times T' \times \ldots \to T_0$, where we "freeze" the powerset parameter to a constant type and we add an extra parameter T'. Intuitively, the method m_1 accounts for the behaviour of m when the powerset parameter is the empty set, while the method m_2 accounts for the case of non-empty sets (the precise definition of $m_1,\ m_2$ will be given in Definition 5.3.1 below).

By applying the above transformations to a binary method, we get a (possibly infinite) set of simple binary methods $m : X \times \prod_{j \in J} T_j \to T_0$, where J is a possibly infinite set of indices (if all sum and product types in the original method are finite, then the number of simple binary methods together with their parameters are finite).

In the second step, we reduce each simple binary method to a bunch of *unary* methods. Let $m : X \times \prod_{j \in J} T_j \to T_0$ be a simple binary method implemented by the function α. In order to capture the observable behaviour of the method m, we need to consider a bunch of unary methods m_l, one for each class parameter, where m_l describes the behaviour of an object when it is used as l^{th} class parameter.

Formally, step 1 and step 2 are defined in terms of the following method transformation:

Definition 5.3.1 (Method/Class Transformation)
(i). Let τ_F be the one-step method transformation *function, which takes a method $m : X \times \prod_{j=1}^{q} T_j \to T_0$, implemented by α, and produces a set of methods, defined by induction on types of m as follows:*

- *if m is* simple binary*, then let I be the set of indices corresponding to the class parameters*

of type X in m, we define

$$\tau_F(m) = \{m_l \mid l \in I\}$$

where $m_l : X \times \prod_{j=1}^{q}(T_j[\bar{X}/X]) \to T_0$ is defined by

$$\alpha_l(x)(a_1, \ldots, a_q) \triangleq \alpha(a_l)(a_1, \ldots, a_{l-1}, x, a_{l+1}, \ldots, a_q) \ .$$

- *if m is non-simple generalized binary and its leftmost non-constant parameter different from X is T_i, then*
  - *if $T_i = \sum_{j=1}^{q_i} T_{ij}$, then $\tau_F(m) = \{m_{ij} \mid j = 1, \ldots, q_i\}$ and $m_{ij} : X \times \ldots \times T_{i-1} \times T_{ij} \times T_{i+1} \times \ldots \times T_q \to T_0$ is defined by*

    $$\alpha_{ij}(x)(a_1, \ldots, a_{i-1}, a_{ij}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, in_j(a_{ij}), \ldots, a_q),$$

    where $in_j : T_{ij} \to \sum_{j=1}^{q_i} T_{ij}$ is the canonical injection.
  - *if $T_i = \prod_{j=1}^{q_i} T_{ij}$, then $\tau_F(m) = \{m'\}$, where $m' : X \times \ldots \times T_{i-1} \times T_{i1} \times \ldots \times T_{iq_i} \times \ldots T_q \to T_0$ is defined by*

    $$\alpha'(x)(a_1, \ldots, a_{i-1}, a_{i1}, \ldots, a_{iq_i}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, \vec{a}_i, a_{i+1}, \ldots a_q).$$

  - *if $T_i = \mathcal{P}(T_i')$, then $\tau_F(m) = \{m_1, m_2\}$, where $m_1 : X \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_q \to T_0$ is defined by $\alpha_1(x)(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, \phi, a_{i+1}, \ldots a_q)$ and $m_2 : X \times \ldots \times T_{i-1} \times \mathcal{P}(T_i'[\bar{X}|X]) \times X \times T_{i+1} \times \ldots \times T_q \to T_0$ is defined by $\alpha_2(x)(a_1, \ldots, a_{i-1}, u, y, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1 \ldots a_{i-1}, u \cup \{y\}, a_{i+1}, \ldots, a_q)$.*

(ii). Let τ_F^* be the transformation function which takes a method and iteratively applies τ_F, defined by

$$\tau_F^*(m) = \begin{cases} \tau(m) & \text{if } m \text{ is simple binary} \\ \bigcup\{\tau_F^*(m_i) | m_i \in \tau_F(m)\} & \text{otherwise} \end{cases}$$

(iii). Finally, for a class C, let C^* be the class with same carrier, fields and constructors of C, and with unary methods $\bigcup\{\tau_F^*(m) | m$ is a method of $C\}$.

Here we apply the above method transformation to a given class:

Example 5.3.2 *Let R' be a class of registers with carrier $\mathbb{N}$, including the method $m : X \times \mathcal{P}_f X \to \mathbb{B}$ defined by:*

$$\alpha(x)(u) = \begin{cases} true & \text{if } x \in u \\ false & \text{otherwise} \end{cases}$$

Then $\tau_F^*(m) = \{\tau_F^*(m_1), \tau_F^*(m_2)\}$, where

- *$m_1 : X \to \mathbb{B}$ is defined by $\alpha_1(x) = \alpha(x)(\phi) = false$*
- *$m_2 : X \times \mathcal{P}_f(\mathbb{N}) \times X \to \mathbb{B}$ is defined by $\alpha_2(x)(u, y) = \alpha(x)(u \cup \{y\})$*
- *$\tau_F^*(m_1) = \{m_1\}$*
- *$\tau_F^*(m_2) = \{m_2', m_2''\}$, where*
  - *$m_2' : X \times \mathcal{P}_f(\mathbb{N}) \times \mathbb{N} \to \mathbb{B}$ is defined by $\alpha_2' = \alpha_2$*
  - *$m_2'' : X \times \mathcal{P}_f(\mathbb{N}) \times \mathbb{N} \to \mathbb{B}$ is defined by $\alpha_2''(x)(u, y) = \alpha_2(y)(u, x) = \alpha(y)(u \cup \{x\})$.*

■

Now, given a class C, we can define a coalgebraic model of the transformed class C^* using purely covariant tools, as in Section 5.2 for unary methods.

Definition 5.3.3 (Freezing Coalgebraic Model) *Let C be a class and let C^* be the class obtained from the transformation τ^* starting from C. We call* freezing functor *F the functor induced by the methods of C^* according to Definition 5.2.1, and* freezing equivalence *$\approx_F$ the bisimilarity equivalence induced by this coalgebraic model.*

Finally, we are left to establish the result which motivates our treatment, namely, for *finitary* binary methods, we can prove that the freezing bisimilarity equivalence is the greatest congruence *w.r.t.* methods of the *original class* C.

Theorem 5.3.4 *Let C be a class with finitary binary methods. Then the freezing bisimilarity equivalence $\approx_F$ is the greatest congruence on C.*

Proof. The proof follows from the following facts:

1. $\approx_F$ is the greatest congruence *w.r.t.* methods in C^*.
2. Any equivalence on objects of C is a congruence *w.r.t.* the methods of C *iff* it is a congruence *w.r.t.* the methods of C^* .

Fact 1 above is immediate by Lemma 5.2.7. In order to prove fact 2 above, it is sufficient to show that an equivalence $\sim$ on a set of objects $\bar{X}$ is a congruence *w.r.t.* a method $m : X \times \prod_{j\in J} T_j \to T_0$ *iff* it is a congruence *w.r.t.* the methods in $\tau_F(m)$. This latter fact is proved by induction on the structure of the parameters $\prod_{j\in J} T_j$.
Base Case: m is a simple binary method implemented by α. If $\sim$ is a congruence *w.r.t.* m, then $\sim$ is immediately a congruence *w.r.t.* the methods in $\tau_F(m)$, by definition. Vice versa, assume that $\sim$ is a congruence *w.r.t.* the methods in $\tau_F(m)$. Let $xa_1 \dots a_q, x'a'_1 \dots a'_q \in X \times \prod_{j=1}^{q} T_j$ be such that $xa_1 \dots a_q \sim x'a'_1 \dots a'_q$. We prove that $\alpha(x)(\vec{a}) \sim \alpha(x')(\vec{a}')$ by induction on the number n of different parameters in the lists $xa_1 \dots a_q, x'a'_1 \dots a'_q$. If $n = 0$, then the thesis is immediate from reflexivity of $\sim$. Let us assume that the thesis holds for $n-1$ different parameters. Now assume that $xa_1 \dots a_q$ and $x'a'_1 \dots a'_q$ have $n > 0$ different parameters, and let a_j, a'_j be the n^{th} different parameters. By induction hypothesis, $\alpha(x)(a_1, \dots, a_j, \dots a_q) \sim \alpha(x')(a'_1, \dots, a_j, \dots a'_q)$. Moreover, $\alpha(x')(a'_1, \dots, a_j, \dots a'_q) \sim \alpha(x')(a'_1, \dots, a'_j, \dots a'_q)$, by the hypothesis that $\sim$ is a congruence *w.r.t.* $\tau_F(m)$. Hence, by transitivity of $\sim$, we get the thesis.
Induction step: If the leftmost non-constant parameter in m different from X is of the shape $\sum_{k\in K} T_{i_K}$ or $\prod_{k\in K} T_{i_K}$, then the thesis is immediate from the definition of $\tau_F(m)$ and that of relational lifting. If the leftmost non-constant parameter in m different from X is $T_i = \mathcal{P}_f(T'_i)$, then $\tau_F(m) = \{m_1, m_2\}$, where $m_1 : X \times \ldots \times T_{i-1} \times T_{i+1} \times \ldots \times T_q \to T_0$, $m_2 : X \times \ldots \times T_{i-1} \times \mathcal{P}_f(T'_i[\bar{X}|X]) \times X \times T_{i+1} \times \ldots \times T_q \to T_0$. Assume that $\sim$ is a congruence *w.r.t.* m, *i.e.*

$$x \sim x' \wedge a_1 \sim a'_1 \wedge \ldots \wedge a_{i-1} \sim a'_{i-1} \wedge u \sim u' \wedge a_{i+1} \sim a'_{i+1} \wedge \ldots \wedge a_q \sim a'_q \Longrightarrow$$

$$\alpha(x)(a_1, \dots, a_{i-1}, u, a_{i+1}, \dots, a_q) \sim \alpha(x')(a'_1, \dots, a'_{i-1}, u', a'_{i+1}, \dots, a'_q)$$

Then in particular $\sim$ is a congruence *w.r.t.* both m_1 and m_2.
Vice versa assume that $\sim$ is a congruence *w.r.t.* m_1, m_2, *i.e.*

$$\begin{aligned} &x \sim x' \wedge a_1 \sim a'_1 \wedge \ldots \wedge a_{i-1} \sim a'_{i-1} \wedge a_{i+1} \sim a'_{i+1} \wedge \ldots \wedge a_q \sim a'_q \Longrightarrow \\ &\alpha(x)(a_1, \dots, a_{i-1}, \emptyset, a_{i+1}, \dots, a_q) \sim \alpha(x')(a'_1, \dots, a'_{i-1}, \emptyset, a'_{i+1}, \dots, a'_q) \end{aligned} \tag{5.3.1}$$

and for all u,

$$\begin{aligned} &x \sim x' \wedge a_1 \sim a'_1 \wedge \ldots \wedge a_{i-1} \sim a'_{i-1} \wedge y \sim y' \wedge a_{i+1} \sim a'_{i+1} \wedge \ldots \\ &\wedge a_q \sim a'_q \Longrightarrow \alpha(x)(a_1, \dots, a_{i-1}, u \cup \{y\}, a_{i+1}, \dots, a_q) \sim \\ &\qquad \alpha(x')(a'_1, \dots, a'_{i-1}, u \cup \{y'\}, a'_{i+1}, \dots, a'_q) \end{aligned} \tag{5.3.2}$$

Now let $x \sim x' \wedge a_1 \sim a'_1 \wedge \ldots \wedge a_{i-1} \sim a'_{i-1} \wedge u \sim u' \wedge a_{i+1} \sim a'_{i+1} \wedge \ldots \wedge a_q \sim a'_q$. We have to show that $\alpha(x)(a_1, \dots, a_{i-1}, u, a_{i+1}, \dots, a_q) \sim \alpha(x')(a'_1, \dots, a'_{i-1}, u', a'_{i+1}, \dots, a'_q)$.

We proceed by induction on the number of elements in $(u \setminus u') \cup (u' \setminus u)$.
If $u = u' = \phi$ then the thesis follows immediately by (5.3.1), if $u = u' \neq \phi$, then the thesis follows by (5.3.2).
If $|u \setminus u'| > 0$, then let $y \in u \setminus u'$. Since $u \sim u'$, there exists $y' \in u$ such that $y \sim y'$. By (5.3.2) we have $\alpha(x)(a_1, \ldots, a_{i-1}, u, a_{i+1}, \ldots, a_q) \sim \alpha(x')\ (a'_1, \ldots, a'_{i-1}, (u \setminus \{y\}) \cup \{y'\}, a'_{i+1}, \ldots, a'_q)$. Now, by definition of relational lifting, using the fact that $\sim$ is an equivalence, we get $(u \setminus \{y\}) \cup \{y'\} \sim u'$. Then, by induction hypothesis, $\alpha(x')(a'_1, \ldots, a'_{i-1}, (u \setminus \{y\}) \cup \{y'\}, a'_{i+1}, \ldots, a'_q) \sim \alpha(x')(a'_1, \ldots, a'_{i-1}, u', a'_{i+1}, \ldots, a'_q)$. Then the thesis follows by transitivity of $\sim$. ∎

As a by-product of Theorem 5.3.4 above, we get that for finitary binary methods the greatest congruence *w.r.t.* methods always exists, *i.e.*

Corollary 5.3.5 *Let C be a class with carrier X and whose methods are all finitary binary. Then $\cup\{\sim \subseteq X \times X \mid \sim$ is a congruence w.r.t. the methods of $C\}$ is a congruence.*

Notice that, in order to ensure Theorem 5.3.4 above, it is essential to give a coalgebraic description of binary methods which accounts for the behaviour of an object under method application when the object is viewed as *any* of the class parameters of the method. Otherwise, if we observe the behaviour of an object *e.g.* only when it is considered as the target of a method call and not as a generic class parameter, the congruence property of $\approx_F$ fails, in general. The following is a counterexample.

Example 5.3.6 *Let us consider a class R' of registers containing just a method $m : X \times X \to \mathbb{N}$, defined by $\alpha(r_1)(r_2) = r_2.val$.*

Now, if in the definition of the freezing functor F we consider only the first component induced by the method m, we have $r_1 \approx_F r_2$, for all r_1, r_2. But then $\approx_F$ is not a congruence w.r.t. the method m. E.g., if we consider r_1, r_2 such that $r_1.val = 1$ and $r_2.val = 2$, then $r_1 \approx_F r_2$, however, for any r_0, $\alpha(r_0)(r_1) = 1$, while $\alpha(r_0)(r_2) = 2$. The problem arises since the result of applying m depends on an unobservable behaviour of the second parameter. ∎

Nevertheless, there are many interesting cases in which it is sufficient to consider only some components in the definition of F for the bisimilarity equivalence to be a congruence. An interesting example is that of the λ-calculus, [41]. In this case, the freezing functor with only first component for the method *app* induces the *applicative equivalence* on closed λ-terms. While, if we consider both components in the functor (or only the second one), we get an equivalence which can be viewed as a coinductive characterization of the *contextual equivalence*. Applicative and contextual equivalences can be proved to coincide. This is not immediate and many techniques, which apply to various reduction strategies, have been developed to achieve this aim, *e.g.* see [41] for more details.

Infinitary Binary Methods.

Theorem 5.3.4 above does not extend to infinitary binary methods, since the freezing bisimulation equivalence fails, in general, to be a congruence. The following are two counterexamples.

Example 5.3.7 *Let C be a class with carrier $X \triangleq \mathbb{N}$ and with just one method $m : X \times \mathcal{P}(X) \to \mathbb{B}$ defined by:*

$$\alpha(x)\mu(U) = \begin{cases} true & \text{if } |\mu(U)| < \omega \\ false & \text{otherwise} \end{cases}$$

One can check that the equivalence $\sim_k = \{(n,m) | n, m \leq k\} \cup \{(n,n) | n > k\}$ is a congruence for all k. However, $\bigcup_{k \in \omega} \sim_k = \{(n,m) | n, m \in N\}$, which coincides with the freezing equivalence, is not *a congruence.* ∎

The following example amounts to Example 3.5.10 of [77], page 114.

Example 5.3.8 *Let C be a class with carrier $X \triangleq \mathbb{N}$ and with just one method $m : X \times [\mathbb{N} \to X] \to \mathbb{B}$ (where $[\mathbb{N} \to X]$ is an alias for the infinite product type $\prod_{i\in\mathbb{N}} X^i$), defined by:*

$$\alpha(x)(f) = \begin{cases} true & \text{if } f \text{ is bounded} \\ false & \text{otherwise} \end{cases}$$

where $f : \mathbb{N} \to \mathbb{N}$ is bounded if there exists $k \in \mathbb{N}$ such that $\forall n. f(n) \leq k$.

One can check that the equivalence $\sim_k = \{(n, m) | n, m \leq k\} \cup \{(n, n) | n > k\}$ is a congruence for all k. However, $\bigcup_{k\in\omega} \sim_k = \{(n, m) | n, m \in \mathbb{N}\}$ coincides with the freezing equivalence and is not *a congruence.* ■

Clearly, in all cases where the union of all congruences is *not* a congruence itself, the freezing equivalence *cannot* be a congruence, since any congruence is in particular a freezing bisimulation. This is not surprising, since in these cases we lack a canonical congruence, thus any semantics would be problematic. A natural question which arises is whether also the other implication holds, *i.e.* if the union all congruences is a congruence, then the freezing equivalence is a congruence. Somehow surprisingly, this is not the case, the following being a counterexample:

Example 5.3.9 *Let C be a class with carrier $X \triangleq \mathbb{N}$ and with just one method $m : X \times [\mathbb{N} \to X] \to \mathbb{B}$ defined by:*

$$\alpha(x)(f) = \begin{cases} true & \text{if } f \text{ is definitely constantly 0} \\ false & \text{otherwise} \end{cases}$$

where $f : \mathbb{N} \to \mathbb{N}$ is definitely constantly 0 if there exists k such that $f(n) = 0$ for all $n \geq k$.

One can easily check that the greatest congruence on X is $\{(0, 0)\} \cup \{(n, m) |\ n, m \neq 0\}$. However, the freezing equivalence is $\{(n, m) | n, m \in \mathbb{N}\}$, which is clearly not a congruence. ■

Nevertheless, there are many situations where the greatest congruence exists and the freezing equivalence captures it. We feel that a situation where the greatest congruence does not exist or it exists but the freezing equivalence does not capture it, is a situation where the class specification is underspecified. But more work needs to be done in order to capture this.

Finally, notice that the problems with infinitary methods arise because of infinite products and $\mathcal{P}(\ )$. Infinite sums are not problematic. Namely, Theorem 5.3.4 above holds also for the extension of finitary types with infinite sums.

5.3.2 The Graph Functor

In this section, we introduce an alternate approach to dealing with binary methods, which is satisfactory in most cases, and when it does not, it is a telltale that the specification is probably under determined.

Contravariant occurrences of the type variable in a generalized binary method can be turned into covariant ones by interpreting methods as *graphs* instead of functions. Consequently, the function space appearing in the functor induced by m is turned into a set of relations. For example, for the binary method $eq : X \times X \to \mathbb{B}$ of the class R of registers, we would consider the functor $GX \triangleq \mathcal{P}(X \times \mathbb{B})$.

Similarly to the case of freezing, in order to make the graph bisimilarity equivalence a congruence in a wider spectrum of cases (including Example 5.3.6), we need to consider multiple copies of the binary method in the definition of the graph functor, in order to account for the behaviour of each class parameter. Thus, the graph functor corresponding to the method *eq* becomes $GX \triangleq \mathcal{P}(X \times \mathbb{B}) \times \mathcal{P}(X \times \mathbb{B})$. This works straightforwardly for simple binary methods, but it requires a preliminary transformation for methods with more complex class parameters. This essentially corresponds to the method transformation procedure for the freezing functor, apart from the part which actually freezes the parameters. For dealing with the powerset type constructor, we introduce the new symbol $\mathcal{P}^{\surd}$, which is used to denote a powerset type constructor which has been already processed.

Formally, we define:

Definition 5.3.10 (Normal Form) *A binary method $m : X \times \prod_{j\in J} T_j \to T_0$ is in* normal form *if its parameter types are either constants or X or $\mathcal{P}^{\surd}(T)$, for $T \in \mathcal{T}$.*

Definition 5.3.11 (Graph Method/Class Transformation)
(i). Let τ_G be the one-step method transformation function, which takes a generalized binary method $m : X \times \prod_{j=1}^{q} T_j \to T$ implemented by α and produces a set of methods, defined by:

- *if m is in normal form, then let I be the set of indices corresponding to class parameters of type X including the object itself, we define*

$$\tau_F(m) = \{m_l \mid l \in I\}$$

where $m_l : X \times \prod_{j\in J} T_j \to T_0$ is defined by

$$\alpha_l(x)(a_1, \ldots, a_q) \triangleq \alpha(a_l)(a_1, \ldots, a_{l-1}, x, a_{l+1}, \ldots, a_q) \ .$$

- *if m is not in normal form, let T_i be the leftmost parameter not in normal form, then*
  - *if $T_i = \sum_{j=1}^{q_i} T_{ij}$, then $\tau_G(m) = \{m_{ij} \mid j = 1, \ldots, q_i\}$ where $m_{ij} : X \times \ldots \times T_{i-1} \times T_{ij} \times T_{i+1} \times \ldots \to T_0$ is defined by $\alpha_{ij}(x)(a_1, \ldots, a_{i-1}, a_{ij}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots a_{i-1}, in_j(a_{ij}), \ldots a_q)$.*
  - *if $T_i = \prod_{j=1}^{q_i} T_{ij}$, then $\tau_G(m) = \{m'\}$, where $m' : X \times \ldots \times T_{i1} \times \ldots \times T_{iq_i} \times \ldots \to T_0$ is defined by $\alpha'(x)(a_1, \ldots, a_{i-1}, a_{i1}, \ldots, a_{iq_i}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, \vec{a}_i, a_{i+1}, \ldots a_q)$.*
  - *if $T_i = \mathcal{P}(T_i')$, then $\tau_G(m) = \{m_1, m_2\}$, where $m_1 : X \times \ldots \times T_{i-1} \times T_{i+1} \times \times \ldots \to T_0$, $\alpha_1(x)(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots, a_{i-1}, \phi, a_{i+1}, \ldots a_q)$ and $m_2 : \ldots \times \mathcal{P}^{\surd}(T_i') \times X \times \ldots \to T_0$ is defined by $\alpha_2(x)(a_1, \ldots, a_{i-1}, u, y, a_{i+1}, \ldots, a_q) \triangleq \alpha(x)(a_1, \ldots a_{i-1}, u \cup \{y\}, a_{i+1}, \ldots a_q)$.*

(ii). Let τ_G^* be the transformation function which takes a generalized binary method and produces a set of methods in normal form, defined by

$$\tau_G^*(m) = \begin{cases} \tau_G(m) & \text{if } m \text{ is in normal form} \\ \bigcup\{\tau_G^*(m_i) | m_i \in \tau_G(m)\} & \text{otherwise.} \end{cases}$$

(iii). Let S be a class specification, we denote by S^* the class specification obtained by applying the transformation τ_G^* to all method declarations in S.
iv) Let C be a class implementation, we denote by C^* the class implementation obtained by applying the transformation τ_G^* to all methods in C.

Notice that there is a precise correspondence between the transformations τ_G^* and τ_F^*. Namely, for any method m, there is a one-to-one correspondence between the methods of $\tau_G^*(m)$ and $\tau_F^*(m)$, mapping each method of $\tau_G^*(m)$ into a method of $\tau_F^*(m)$, which differs from the first one only because of the freezing of some parameter types.

In order to give a graph coalgebraic model to a specification S (implementation C), we consider the corresponding transformed specification S^* (implementation C^*), and we define a corresponding graph functor G, simply by turning the contravariant function spaces in method declarations into covariant spaces of graphs:

Definition 5.3.12 (Graph Functor) *Let S be a class specification. The method declarations in S^* induce the* graph functor *$G : \mathcal{C} \to \mathcal{C}$ defined by*

$$G \triangleq \prod_{i=1}^{k} G_i \ ,$$

where, for each method $m_i : X \times \prod_{j\in J} T_{ij} \to T_{i0}$, $G_i X \triangleq \mathcal{P}((\prod_{j\in J} T_{ij}) \times T_{i0})$.

Given a class C, the corresponding class C^* immediately induces a coalgebra for the graph functor determined by the method declarations in C^*, according to Definition 5.2.2 of Section 5.2, by viewing method codes as graphs instead of functions. However, there is not a precise correspondence between classes and coalgebras anymore, since not all G-coalgebras correspond to a class, but only the functional ones, *i.e.* those whose coalgebra map is a function.

The graph bisimilarity equivalence on the objects of C^* can be characterized as follows:

Proposition 5.3.13 (Graph Bisimilarity Equivalence) *Let C be a class, let $G = \prod_{i=1}^{k} G_i$ be the functor induced by the method declarations in C^*, and let $(X, \langle \alpha_i \rangle_{i=1}^{k})$ be the G-coalgebra induced by the methods of C^*. Then*

(i). A G-bisimulation (graph bisimulation) on $(X, \langle \alpha_i \rangle_{i=1}^{k})$ is a relation $R \subseteq X \times X$ satisfying

$$x\,R\,x' \Longrightarrow \forall \alpha_i.\, \forall \vec{a}\, \exists \vec{a}'.\, (\vec{a}\,R\,\vec{a}' \wedge\ \alpha_i(x)(\vec{a})\,R\,\alpha_i(x')(\vec{a}')) \wedge$$
$$\forall \vec{a'}\ \exists \vec{a}.(\vec{a}\,R\,\vec{a'}\ \wedge \alpha_i(x)(\vec{a})\,R\,\alpha_i(x')(\vec{a'}))\ .$$

(ii). The graph bisimilarity equivalence $\approx_G$, i.e. the greatest G-bisimulation on $(X, \langle \alpha_i \rangle_{i=1}^{k})$, can be characterized as follows:

$$x \approx_G x' \iff \forall \alpha_i.\ \forall \vec{a}\ \exists \vec{a'}.(\vec{a} \approx_G \vec{a}'\ \wedge \alpha_i(x)(\vec{a}) \approx_G \alpha_i(x')(\vec{a}')) \wedge$$
$$\forall \vec{a'}\ \exists \vec{a}.(\vec{a} \approx_G \vec{a'}\ \wedge \alpha_i(x)(\vec{a}) \approx_G \alpha_i(x')(\vec{a'}))\ .$$

In particular, the following coinduction principle holds:

$$\frac{R \text{ is a graph bisimulation} \qquad x\,R\,x'}{x \approx_G x'}$$

Proof. By definition of coalgebraic bisimulation (see Definition 3.4.2 of Chapter 3). ∎

Notice the alternation of quantifiers $\forall\exists$ in the definition of graph bisimulation, due to the presence of the powerset in the graph functor.

The functor G has a final coalgebra, see *e.g.* [17]. But, in general, it is not functional, and moreover the functionality property of a coalgebra is not preserved by the unique morphism into the final coalgebra. Therefore, the image of a class implementation under the final morphism is not guaranteed to be a class implementation. Thus we lack minimal class implementations, in general. In Section 5.3.3, we study conditions for the final morphism to preserve the functionality property, thus recovering minimal implementations.

5.3.3 Comparing Graph and Freezing Bisimilarity Equivalences

The following is an easy lemma:

Lemma 5.3.14 $\approx_F \subseteq \approx_G$.

Proof. One can easily check that $\approx_F$ is a graph bisimulation, using reflexivity of $\approx_F$. ∎

The converse inclusion does not hold in general. For example, this is the case for the class R' obtained from the class R of registers of Table 4.1 when we drop methods *get* and *set*, and we consider only method *eq*. Namely, for R', $\approx_G$ equates all pairs of registers, while $\approx_F$ is the identity relation on registers. Moreover, notice that in this case $\approx_G$ is *not* a congruence *w.r.t. eq*.

The following result is a fundamental tool for recovering $\approx_F = \approx_G$:

Theorem 5.3.15 *Let C be a class with finitary generalized binary methods. Then*

$$\approx_G = \approx_F \iff \approx_G \text{ is a congruence w.r.t. the methods in the class.}$$

Proof. ($\Rightarrow$) By Theorem 5.3.4.
($\Leftarrow$) By Lemma 5.3.14, $\approx_F \subseteq \approx_G$. Since $\approx_G$ is a congruence, by Theorem 5.3.4, $\approx_G \subseteq \approx_F$. Thus $\approx_G = \approx_F$. ∎

The equality $\approx_G = \approx_F$ on a given class C is equivalent to the fact that the image of the G-coalgebra representing C into the final coalgebra is still a functional coalgebra. Hence, we have:

Corollary 5.3.16 *Let C be a class with finitary generalized binary methods. Then the image of the G-coalgebra representing C into the final coalgebra is a functional coalgebra* if and only if $\approx_G$ *is a congruence.*

Thus Corollary 5.3.16 above gives an answer to the problem of minimal class implementations for the graph functor, raised at the end of Section 5.3.2.

Theorem 5.3.15 above is all that we might want. However, in practice, it is useful to have also alternative sufficient conditions. The following theorem gives a sufficient condition on the freezing equivalence, ensuring that $\approx_G = \approx_F$:

Theorem 5.3.17 *If $\approx_F$ is determined only by the unary methods of the class, i.e.*

$$x \approx_F x' \iff \forall m \text{ unary implemented by } \alpha.\ \forall \vec{a}.\ \alpha(x)(\vec{a}) \approx_F \alpha(x')(\vec{a'}) \ ,$$

then $\approx_G = \approx_F$.

Proof. By Lemma 5.3.14, $\approx_F \subseteq \approx_G$. Vice versa, we have

$$\approx_F = (\approx_F)_{|\text{unary methods}} = (\approx_G)_{|\text{ unary methods}} \supseteq \approx_G .$$

∎

Theorem 5.3.17 above applies to the class R of registers, since the freezing equivalence is already determined solely by the unary methods *get* and *set*.

Remark 5.3.18 Notice that Theorem 5.3.17 does not apply to the class Λ, where nevertheless $\approx_G = \approx_F$. Proving this latter result for the λ-calculus is quite a difficult task. This problem has been addressed in the more general setting of applicative structures in [42]. ∎

An almost trivial, but useful application of Theorem 5.3.17 is the following:

Corollary 5.3.19 *If the freezing equivalence restricted to the unary methods of the class is the identity on objects, then $\approx_G = \approx_F$.*

5.4 Remarks and Directions for Future Work

We end this chapter with some comments and some potentially fruitful lines of research about coalgebraic semantics of binary methods.

- Without the powerset, our (finitary) generalized binary methods are a subset of the ones handled by Tews using extended polynomial (cartesian) functors. However, we feel that the two collections of methods essentially correspond. Namely, given a method m which has an extended polynomial type, either m corresponds directly (possibly up-to currying) to a generalized binary method, or m can be cast into a generalized binary type at the price of extending it vacuously. For example, a method $m : X \to ((X \to (N \to X)) + N)$ has an extended polynomial type in the sense of Tews, but not a generalized binary type (since the occurrence of $+$ prevents currying). But, the effects of m can be recovered in our setting, since m can be cast into a method of type $X \times X \to ((N \to X) + N)$. Of course, this is just an example, and further investigation is needed to streamline this procedure.

- Contravariant domain equations appropriate for expressing directly binary methods can be solved both in the category of CPO's and metric spaces [10, 70]. The problem arises then as to which equivalence is induced by these semantics. In the coalgebraic approaches, that we discuss in the thesis, the equivalence is *always* the greatest congruence.
  - In the CPO the metric cases, one needs to prove that the semantics is fully abstract, in order to establish this.
  - It would be very interesting to investigate this problem and possibly prove a general full abstraction result. Such a result would shed light on the mathematical structure of CPO's (or metric spaces).

  Furthermore, it would provide a new induction, or a new coinduction principle.

- The existence of final (minimal) models for a given specification is important. To this aim, as discussed at the end of Section 5.2, it is crucial that the bisimilarity equivalence is a congruence and moreover it preserves assertions. It would be quite interesting to investigate for which kinds of assertions this is the case.

- Following [47], one can also define an equivalence between coalgebras implementing the same specification, by taking coalgebras to be equivalent when initial objects are bisimilar.

- The grammar for parameter types in Definition 5.1.1 could be extended to include inductive and coinductive types. However, apparently, it cannot be extended with the (contravariant) arrow type. Namely, there is no "well-behaved" natural extension of the behavioural equivalence to the function type, since the natural definition $R^{T_1 \to T_2} = \{(f, f') \mid \forall x R^{T_1} x'.\ f x R^{T_2} f' x'\}$ of relational lifting fails to preserve equivalence relations, because $R^{T_1 \to T_2}$ is not reflexive, in general. A possible remedy to this problem is that of including $T \to T$ in the *covariant* space of *binary relations*. The difference *w.r.t.* the traditional interpretation of the function space arises when we define bisimulation equivalences.

- Finally, we point out that our approach to the bialgebraic description of classes involving binary methods is quite general. It can be used to model coinductive data types, possibly with binary evolution laws, such as the concurrent process language with process parameters studied in [51].

6

Towards Co(bi)algebraic Descriptions of Object-Oriented Languages with Store

In this chapter, we investigate the possibility of extending the coalgebraic approach discussed in Chapter 5 to imperative Object-Oriented languages. We focus on a fragment of the imperative typed class-based language *Fickle* [29, 30], introduced in Chapter 2.

In dealing with Fickle, the approach investigated in Chapter 5 needs to be refined to accommodate *imperative* features as well as general *programs*, i.e. sequences of classes possibly related by inheritance, mutual definitions, etc. Special care needs to be devoted to representing the *store*, and in defining the evolution of objects, we have to take into account all possible pointers involving them.

In this chapter, we deal directly only with *unary* methods, and we point out at the extra problematic issues which arise in the imperative setting, when we try to model also binary methods, in the line of Chapter 5.

Moreover, in this chapter, we investigate the possibility of utilizing the coalgebraic computational model also for *program equivalence* and *program transformation*. This is a somewhat dual goal w.r.t. the program refinement of Reichel and Jacobs.

Interestingly, the coalgebraic equivalence on Fickle objects induces a behavioural equivalence on Fickle expressions (*i.e.* bodies of *main* methods), which can be used to study notions of *observational equivalences*. In particular, in this chapter we use the coalgebraic equivalence to study the *contextual equivalence* $\approx_P$ (indexed by a program P), introduced in Chapter 2, Definition 2.3.1.

6.1 Coalgebraic Description of Fickle Objects and Programs

In this section, we give a coalgebraic account of Fickle objects (and programs) for the fragment of Fickle consisting of *unary methods*. Following the approach in Chapter 5, we model classes as coalgebras, where the carrier represents the objects of the classes, and the coalgebra structure is determined by the operational semantics of the methods. The coalgebra structure captures the evolution of the objects under the action of methods. In order to model the evolution of objects in an imperative setting, we need to account also for sharing of addresses in the store and aliasing of variables.

We start by defining our representation of imperative objects of a class c.

Definition 6.1.1 *Let* refobject$_c$ *be the set of pairs* (ι, O)*, where* $\iota \in addr$*, and* $O \in (addr \rightarrow_{pfin} object)$ *is the least* closed *function, — i.e.* $\forall o \in range(O).\ \forall \iota \in range(o).\ \iota \in dom(O)$ *—, such that* $\iota \in dom(O)$ *and* $O(\iota) \in object_c$*.*

Essentially, a refobject can be viewed as a *minimal* store induced by an address, when we do not consider the environment part.

In what follows, we simply denote by O an element (ι, O) of $\bigcup_c refobject_c$.

Before introducing our coalgebraic semantics, we need to define the notion of *consistency* between refobjects and stores, the notion of store update with a refobject, and the notion of refobject induced by an address in a store:

Definition 6.1.2 *Let $\sigma \in store$, $O \in refobject_c$, $\iota \in addr$.*
i) O and σ are consistent*, written $con(O,\sigma)$, if for all addresses $\iota \in dom(O)$, if $\iota \in dom(\sigma)$, then $O(\iota) = \sigma(\iota)$.*
ii) For O and σ consistent, and $x \in id$, we define $\sigma[O/x]$ the store σ in which the object corresponding to the refobject *O has been associated to x, and the rest of the store, if necessary, has been updated according to O.*
iii) Let $\iota \in dom(\sigma)$. We denote by $\overline{\sigma(\iota)}$ the unique refobject (ι, O) included in σ. Let $x \in varid \cup \{this\}$. We denote by $\overline{\sigma(x)}$, $\sigma(x)$ itself, if x has base type, the unique refobject $(\sigma(x), O)$ included in σ, otherwise.

Now we introduce the coalgebraic description of the fragment of Fickle consisting of unary methods. To this aim, we endow the set of refobjects of a given program P with a coalgebra structure for the functor induced by the methods in P. A method $t_0\ m(t_1x_1, \ldots, t_qx_q)$ in P, when called on an object together with a list of actual parameters, can either terminate (successfully or with an exception/error) producing a possibly modified object, or not terminate. The behaviour of methods on objects determines the coalgebraic structure:

Definition 6.1.3 *Let $P \triangleq c_1, \ldots, c_n$, where $c_i \triangleq \{f_{i1}; \ldots f_{ih_i}; m_{i1}; \ldots; m_{ik_i}\}$.*
i) Let $H : Set \to Set$ be defined by

$$H \triangleq \coprod_i \prod_j H_{ij} \ ,$$

where $H_{ij} : Set \to Set$ is determined by the method declaration

$$t_0\ m_{ij}\ (t_1x_1, \ldots, t_qx_q)\ \{c'_1, \ldots, c'_p\}$$

of the class c_i as follows:

$$H_{ij}X \triangleq [\![t_1]\!] \times \ldots \times [\![t_q]\!] \to (([\![t_0]\!] + dev) \times X + 1) \ ,$$

where, for all $i = 0, \ldots, q$,

$$[\![t_i]\!] = \begin{cases} bool & \text{if } t_i = bool \\ int & \text{if } t_i = int \\ addr & \text{otherwise .} \end{cases}$$

The definition of H_{ij} on arrows is canonical.
ii) Let us denote $\coprod_{c_i} refobject_{c_i}$ simply by $refobject_P$. Let $\alpha_P : refobject_P \to H(refobject_P)$ be defined by

$$\alpha_P \triangleq [\langle \alpha_{ij} \rangle_j]_i \ ,$$

where $\alpha_{ij} : refobject_{c_i} \to H_{ij}(\coprod_{c'_k \in C_{ij}} refobject_{c'_k})$, for C_{ij} the set of classes to which the method m_{ij} can reclassify the object, is defined by:

$$\alpha_{ij}(O) \triangleq \vec{a} \mapsto \begin{cases} (u, \overline{\sigma_1(this)}) & \text{if } e \text{ is the body of } m_{ij} \text{ and} \\ & (e, \emptyset[O/this, \vec{a}/\vec{x}]) \longrightarrow_P (u, \sigma_1) \\ * & \text{otherwise,} \end{cases}$$

where $*$ denotes the only element of 1. Notice that the store $\emptyset[O/this, \vec{a}/\vec{x}]$ is always defined (i.e. there are no consistency problems), since all actual parameters are of base type.
iii) Let $[\![\ ]\!]_P^H : (refobject_P, \alpha_P) \to (\Omega_G, \alpha_{\Omega_H})$ be the coalgebraic semantics, i.e. the unique H-coalgebra morphism into the final *H-coalgebra.*

By applying the general theory of coalgebraic semantics (see Chapter 3), we get the following coinductive characterization of the equivalence induced by $[\![\ ]\!]^H_P$:

Proposition 6.1.4 *The coalgebraic semantics $[\![\ ]\!]^H_P$ induces the following behavioural equivalence on objects of P: for all $O, O' \in refobject_c$, where c is a class of P,*

$$O \sim^H_P O' \iff$$

$\forall$ *method* $m(\vec{x})$ *in* c *with body* e, $\forall$ *list of arguments* $\vec{a}$ *for* $\vec{x}$,
$(e, \emptyset[O/this, \vec{a}/\vec{x}]) \longrightarrow_P (u, \sigma_1)$
$\Rightarrow\ (e, \emptyset[O'/this, \vec{a}/\vec{x}]) \longrightarrow_P (u, \sigma'_1)\ \wedge\ \overline{\sigma_1(this)} \sim^H_P \overline{\sigma'_1(this)}$, *and conversely.*

Corollary 6.1.5 $\sim^H_P$ *is the greatest fixed point of the following monotone (w.r.t. subset inclusion) operator on relations on refobjects:*
$\Phi(R) \triangleq \{(O, O') \mid \forall m(\vec{x}) : e$ *in* c, $\forall$ *list of arguments* $\vec{a}$ *for* $\vec{x}$,
$(e, \emptyset[O/this, \vec{a}/\vec{x}]) \longrightarrow_P (u, \sigma_1)$
$\Rightarrow\ (e, \emptyset[O'/this, \vec{a}/\vec{x}]) \longrightarrow_P (u, \sigma'_1)\ \wedge\ \overline{\sigma_1(this)}\ R\ \overline{\sigma'_1(this)}$, *and conversely* $\}$.
In other words, the following coinduction principle for establishing $\sim^H_P$ *is sound and complete:*

$$\frac{ORO'\ \wedge\ R \text{ is a } \Phi\text{-bisimulation}}{O \sim^H_P O'}\ ,$$

where a Φ*-bisimulation* R *is a relation s.t.* $R \subseteq \Phi(R)$.

Example 6.1.6 Let *Register* be a class with just one field containing the integer value of a register, and two methods, *getval* and *setval*. The first method returns the contents of the register, the latter sets the contents to a new value passed as parameter, and returns the new value. One can easily check that the coalgebraic equivalence on objects class *Register* equates two registers if and only if they have the same contents. ■

Example 6.1.7 Let *IntList* be a class representing possibly circular lists of integers. The class *IntList* has two fields, representing the head and the tail of a list, i.e. containing an integer value and a list, respectively, and two methods, returning the head and the tail of a list. Then the coalgebraic equivalence on *IntList* equates two lists if and only if they have the same value in the head and the same address in the tail.

■

Example 6.1.8 In order to recover the extensional equivalence on lists, one can define the class *IntList* by considering just one method, taking an integer n as parameter and returning the value of the n-th element of a list. ■

The coalgebraic equivalence $\sim^H_P$ equates objects which behave in the same way under method application, for all lists of parameters, in the *minimal* store. Actually, store minimality is not relevant. Namely, one can easily show that the behaviour of an object only depends on method parameters, and not on the rest of the store, if we assume that the expression **new** c does not appear in the bodies of class methods. However, we conjecture that the above assumption can be eliminated. Anyway, we feel that this is not a strong assumption, since usually class methods are used to access or modify objects, while creation of new objects is performed in the *main* method.

Moreover, in what follows, we tacitly assume also that, if two objects of a root class d are $\sim^H_P$-equivalent, then their canonical extensions (via re-classification) to objects of a state subclass c are still $\sim^H_P$-equivalent. This means that in the subclass c there are no extra methods which discriminate solely on the basis of the fields in the superclass d. This is quite a natural hypothesis, which is necessary to deal with re-classification.

Thus we have:

Lemma 6.1.9

$$O{\sim}_P^H O' \iff$$

$\forall$ *method* $m(\vec{x})$ *in* c *with body* e, $\forall\sigma.\ con(O,\sigma) \ \wedge\ con(O',\sigma)$,
$(e, \sigma[O/this]) \longrightarrow_P (u, \sigma_1)$
$\Rightarrow\ (e, \sigma[O'/this]) \longrightarrow_P (u, \sigma_1') \ \wedge\ \overline{\sigma_1(this)}{\sim}_P^H\overline{\sigma_1'(this)}$, *and conversely.*

The equivalence $\sim_P^H$ on refobjects naturally induces an equivalence on stores, if we take stores to be equivalent on all variables:

Definition 6.1.10 *We define*

$$\sigma \sim_P^H \sigma' \ \stackrel{\Delta}{\iff}\ \forall x \in id.\ \overline{\sigma(x)} \sim_P^H \overline{\sigma'(x)}\ .$$

Another immediate consequence of the fact that object behaviour only depends on method parameters, is that, if two objects are $\sim_P^H$-equivalent, then they behave in the same way under application of methods on $\sim_P^H$-equivalent parameters:

Lemma 6.1.11

$$O{\sim}_P^H O' \iff$$

$\forall\ m(\vec{x})\ :\ e\ in\ c,\ \forall\sigma,\sigma'.\ \sigma \sim_P^H \sigma' \ \wedge\ con(O,\sigma) \ \wedge\ con(O',\sigma'),\ (e, \sigma[O/this]) \longrightarrow_P (u, \sigma_1) \ \Rightarrow$
$(e, \sigma'[O'/this]) \longrightarrow_P (u, \sigma_1') \ \wedge\ \overline{\sigma_1(this)}{\sim}_P^H\overline{\sigma_1'(this)}$, *and conversely.*

6.1.1 Binary Methods

The freezing and graph approach introduced in Chapter 5 for modelling binary methods are not directly extensible to an imperative setting. Namely, in the present imperative setting, binary methods give rise to the extra issue of possible inconsistencies between the object O and the other object parameters, even in the empty store. In particular, if we consider the natural extension of the coalgebraic semantics of unary methods to binary methods, we get an object equivalence which discriminates on the basis of addresses, both in the case of the freezing functor F and in the case of the graph functor G. Namely, let us focus on freezing, and let us consider the class *Register* of Example 6.1.6, extended with the binary method *add*, which adds the contents of two registers. Then the method *add* tells apart registers with different addresses but equal contents, when we apply it to a register parameter consistent with e.g. the first register but not with the second one. To overcome this problem, one could modify the notion of object equivalence, by testing the behaviour of objects under method application only on parameters consistent with both objects. However, somewhat surprisingly, this is not a transitive relation, in general. A possible solution to the transitivity problem above consists again in compensating the observability deficit of unary methods. However, this deserves further study, and we leave it as an open problem how to give a coalgebraic description of Fickle objects in the general case.

6.2 Coalgebraic and Observational Equivalences on Programs

In this section, we introduce a notion of equivalence on expressions w.r.t. a program P, $\dot{\approx}_P$, which is induced by the coalgebraic equivalence on objects $\sim_P^H$ of Section 6.1, and we briefly discuss the relationships between $\dot{\approx}_P$ and the contextual equivalence $\approx_P$ introduced in Chapter 2, Section 2.3.

From now on we denote the equivalence $\sim_P^H$ simply by $\sim_P$. The coalgebraic equivalence on objects introduced in the previous section naturally induces a notion of equivalence on expressions representing bodies of *main* methods w.r.t. a given program P. Two *main* methods are equivalent w.r.t. P when, for any store, they produce equivalent values and equivalent stores:

Definition 6.2.1 *Let* $\dot{\approx}_P \subseteq expr \times expr$ *be defined by:*

$$e \,\dot{\approx}_P\, e' \iff$$

$\forall \sigma.\ ((e,\sigma) \to_P (u,\sigma_1) \implies (e',\sigma) \to_P (u',\sigma_1') \ \wedge\ \overline{u} \sim_P \overline{u}' \ \wedge\ \sigma_1 \sim_P \sigma_1'),$
and conversely,
where $\overline{u}$ *is* u*, if* $u \in sVal \cup dev$*, and it is* $\overline{\sigma_1(u)}$*, if* $u \in addr$*.*

We conjecture that $\dot{\approx}_P$ is adequate w.r.t. the contextual equivalence $\approx_P$, i.e:

Conjecture 6.2.2 *(Adequacy of* $\dot{\approx}_P$*):* $\dot{\approx}_P \subseteq \approx_P$*.*

Completeness of $\dot{\approx}_P$ trivially fails, because in the contextual equivalence there is no way of observing different addresses generated by **new** expressions. For instance, if the class c in the program P is s.t. only objects with the same address are $\sim_P$-equivalent, then the expressions $e \triangleq x := \mathbf{new}\ c$ and $e' \triangleq x := \mathbf{new}\ c; x := \mathbf{new}\ c$ are *not* $\dot{\sim}_P$-equivalent. However, there is no context separating them, since in the observational equivalence $\approx_P$ we only observe values of base types. Nevertheless, we can still get a completeness result for the restricted set of expressions not containing **new** expressions, under the assumption that in each class of the program there is an observable and modifiable field of base type:

Theorem 6.2.3 *(Completeness of* $\approx_P$*): Let* P *be a program, and let* e, e' *be* main *methods for* P*. If* e, e' *do not contain* ***new*** *expressions, and in each class* c *of* P *there is a field* f *of base type, a method* m_1*, which returns the value of* f*, and a method* m_2*, which sets the field* f *to a value given as parameter, then* $e \approx_P e' \implies e \,\dot{\approx}_P\, e'$ *.*

Proof. Assume by contradiction $e \approx_P e'$, but $e \,\dot{\not\approx}_P\, e'$. The difficult case is when $e \,\dot{\not\approx}_P\, e'$ because $\exists \sigma.\ \underline{(e,\sigma)} \to_P \underline{(\iota,\sigma_1)} \ \wedge\ (e',\sigma) \to_P (\iota',\sigma_1')$, but $\overline{\iota} \not\sim_P \overline{\iota}'$ (if returned values are equivalent, but $\exists x.\ \underline{\sigma_1(x)} \not\sim_P \underline{\sigma_1'(x)}$, then we proceed as in the previous case by considering $C[\ ] \triangleq [\ ]; x$). Let us assume that the objects to which ι, ι' point in the stores σ_1, σ_1' are of class c. If $\sigma_1(\iota).f \neq \sigma_1'(\iota').f$, then the context $C[\ ] \triangleq [\ ].m_2(\ldots)$ tells apart e and e', getting a contradiction. Otherwise, if $\sigma_1(\iota).f = \sigma_1'(\iota').f$, then let z be fresh, and let us consider the store $\sigma[\iota/z]$. Then the context $C[\ ] \triangleq z.m_2(\ldots a \ldots); ([\ ].m_1(\ldots) = z.m_1(\ldots))$, where a is a new value for the field f, tells apart e and e' in the store $\sigma[\iota/z]$. ■

Notice that the assumption of no occurrence of **new** expressions in e, e' is fundamental in the proof above, since the technique of extending the store with a fresh variable would not work in the case the addresses ι, ι' are generated by **new** expressions.

6.3 Remarks and Directions for Future Work

In this chapter, we have extended the coalgebraic framework of [64, 47] to the imperative case. Moreover, we have defined a coalgebraic behavioural equivalence, which we conjecture to be adequate w.r.t. the contextual equivalence, and which is complete under suitable restrictions on syntax.

The two main problems which remain open are:

- to accommodate coalgebraically binary methods in the imperative setting;
- to prove the adequacy result of the coalgebraic behavioural equivalence *w.r.t.* the contextual equivalence.

7

Typing Binary Methods

In this chapter, we discuss the well-known problem of typing binary methods when subclasses are considered as subtypes. In Section 7.1, we recall the problem, in Section 7.2, we discuss some solutions which have been proposed in the literature. Finally, in Section 7.3, we present a new solution, which is based on a new typing system, where one can annotate in the type of an object whether a method is never called on that object. In the conclusion we will briefly comment on the motivations for introducing this new system.

7.1 The Problem of Typing Binary Methods

Typing binary methods is problematic when subclasses are considered as subtypes, *i.e.* an object of a subclass can be passed to a method expecting an object of a superclass. This problem has been extensively studied in the literature, see *e.g.* [15]. Here we briefly illustrate it.

The problem arises when overloading of binary methods in subclasses is admitted. Figure 7.1 shows the declaration of a class of points in a plane, a standard example of a class with a binary method. In the class *Point*, the method *get-x* returns the x-position of the point in the plane, the method *get-y* returns the y-position; the method *eq* is binary. It takes two points as arguments to test for the equality and returns *true*, if they are considered as equal.

Figure 7.2 defines a subclass *ColorPoint* of class *Point*. The method *get-s* returns the color of the point. The implementation of the method *eq* allowing two *ColorPoint* objects to be compared (taking the color into account), overrides the behaviour of *eq* of *Point*. In this chapter, we consider this as running example. The instances of *Point* and *ColorPoint* have the following object types:

$$Point \equiv OT\langle\!\langle \text{get-x} : int;\ \text{get-y} : int;\ \text{eq} : Point \to bool \rangle\!\rangle$$

$$ColorPoint \equiv OT\langle\!\langle \text{get-x} : int;\ \text{get-y} : int;\ \text{get-s} : string;\ \text{eq} : ColorPoint \to bool \rangle\!\rangle$$

```
class Point
   attributes
    xValue : int
    yValue : int
   methods
    pt.get-x = pt.xValue
    pt.get-y = pt.yValue
    pt1.eq(pt2 : Point) =
          if (pt1.get-x = pt2.get-x & pt1.get-y = pt2.get-y)
           then true
           else false
end class
```

Figure 7.1: The class *Point*

```
class ColorPoint extends Point
   attributes
    sValue : int
   methods
    cpt.get-s = cpt.sValue
    cpt1.eq(cpt2 : ColorPoint) =
          if (cpt1.get-x = cpt2.get-x & cpt1.get-y = cpt2.get-y &
                cpt1.get-s = cpt2.get-s)
           then true
           else false
end class
```

Figure 7.2: The class *ColorPoint*

```
class d
   attributes
    ...
   methods
    ...
    pt.breakit( ) =
        nupt : Point
        nupt = new Point(1,2)
        if (pt.eq(nupt))
          then true
          else false
end class
```

Figure 7.3: The Method *breakit*

Informally, a type σ is a subtype of τ, written $\sigma \leq \tau$, if an expression of type σ can be used in any context that expects an expression of type τ. In object-oriented programming, an object type σ is a subtype of the type τ if σ has *more methods* than τ, as any context that expects the object of type τ will not directly use the extra methods of σ and thus no type errors will occur. In fact it is also possible to replace the type of any method by a subtype and still have the resulting object types in the subtype relation. Thus, the general rule is

$$OT\langle\langle m_1 : S_1, \ldots, m_n : S_m, \ldots, m_{n+k} : S_{m+k}\rangle\rangle \leq OT\langle\langle m_1 : T_1, \ldots, m_n : T_m\rangle\rangle$$

(with $k \geq 0$) *iff* for each $i \in \{1..n\}, S_i \leq T_i$.

In object-oriented languages, if subclasses corresponds to subtypes, an object of the subclass *ColorPoint* can be used where an object of the superclass *Point* is expected. But not always subclasses generate subtypes. Namely, methods are typed using the contravariant arrow type. The *contravariant* rule [**?**] for the arrow type states that $\sigma \to \tau \leq \sigma' \to \tau'$ *iff* $\sigma' \leq \sigma$ and $\tau \leq \tau'$. Now, let us consider method *eq* in *ColorPoint*. Its type would be a subtype of the type of *eq* in *Point*, if $ColorPoint \to bool$ is a subtype of $Point \to bool$. By the contravariant rule this requires *Point* to be subtype of *ColorPoint*, which is untrue, because of the contravariance arrow type of subtyping relation. On the other hand, it would be unsafe to have $ColorPoint \leq Point$. Namely, consider the method *breakit* of Figure 7.3; when we call this method with colored point *cp*, then it calls the method *eq* of *ColorPoint*. This code tries to access the color field of *nupt* which does not exist.

7.2 Existing Solutions

In the literature,[20, 18, 15], there are various possible solutions to the problem of typing binary methods. In this section, following Kim Bruce et al. [15], we present various solutions for typing

```
class PointPair
  attributes
    p1 : Point
    p2 : Point
  methods
    p1.eq(p2 : Point) = if (p1.get-x=p2.get-x & p1.get-y=p2.get-y)
                        then true
                        else false
end class

class ColorPointPair
  attributes
    p1 : ColorPoint
    p2 : ColorPoint
  methods
    p1.eq(p2 : ColorPoint) =
      if (p1.get-x=p2.get-x & p1.get-y=p2.get-y & p1.get-s=p2.get-s)
        then true
        else false
end class
```

Figure 7.4: The *PointPair* and *ColorPointPair* classes

binary methods.

The simplest way to solve this problem is to avoid binary methods. Depending on the object oriented language used, which provides, say, also conventional procedural abstraction, such as C++, Object Pascal, one can use functions instead of binary methods. As usual, these functions can be defined outside of classes and can be applied to pair of arguments. In our example, the binary method *eq* can be defined outside the classes *Point* and *ColorPoint*, which do not contain method *eq* anymore.

```
function eqPt(pt1, pt2: Point): bool
  return(pt1.get-x=pt2.get-x & pt1.get-y=pt2.get-y);

function eqCPt(cpt1, cpt2: ColorPoint): bool
  return(cpt1.get-x=cpt2.get-x & cpt1.get-y=cpt2.get-y
                               & cpt1.get-s=cpt2.get-s);
```

The problem of using functions instead methods is the loss of *dynamic dispatch* and causes unnecessary code duplication.

In pure object oriented languages, one can place binary methods outside of the classes on which they operate, by making the two argument objects into a single pair object and invoking the method on the pair. That is, the classes *Point* and *ColorPoint* do not contain the binary method *eq*, and hence *ColorPoint* is subtype of *Point*:

$$Point \equiv OT\langle\!\langle \text{get-x} : int;\ \text{get-y} : int \rangle\!\rangle$$

$$ColorPoint \equiv OT\langle\!\langle \text{get-x} : int;\ \text{get-y} : int;\ \text{get-s} : string \rangle\!\rangle$$

But, we need two new classes, *PointPair* and *ColorPointPair*, where the binary method *eq* of *Point* and *ColorPoint* are defined as unary methods, see Figure 7.4.

Avoiding binary methods is the easiest solution, but sometimes it is important to use binary methods. In such case, Kim Bruce et al. [15], presented two important solutions.

The first solution is by using the concept of matching. One object type *matches* another if the first one has at least the methods of the second one, and the corresponding method types are same, when the type of the class is replaced by *self* or *this*. This relationship is denoted by <#,

one can represent as:

$$OT\langle\langle m_1 : \tau_1, \ldots, m_n : \tau_m\rangle\rangle <\# OT\langle\langle m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\rangle$$

iff $k \leq n$.

For example, one can write the type of *Point* and *ColorPoint* as

$$Point \equiv OT\langle\langle \text{get-x} : int;\ \text{get-y} : int;\ \text{eq} : this \to bool\rangle\rangle$$

$$ColorPoint \equiv OT\langle\langle \text{get-x} : int;\ \text{get-y} : int;\ \text{get-s} : string;\ \text{eq} : this \to bool\rangle\rangle$$

this, when used in the definition of the subclass *ColorPoint* automatically represents *ColorPoint* rather than *Point*. Hence, *ColorPoint* <#*Point*. But, the type *ColorPoint* is not a subtype of *Point*. Thus, matching is not subtyping and hence if $S <\# T$, matching does not allow the use of a parameter of type S where type T is expected. Matching tells what parameters can be passed to an object, and what their types will be. We remark that in binary method, we have the same type of arguments (*i.e.* the receiver and the passing parameter). Example 7.2.1 shows the classes *Point* and *ColorPoint* in *OCaml*. See the method *eq* is forbidden, when called with different type of arguments.

Example 7.2.1
```
# class Point (x,y) =
    object (self: 'a)
      val r1 : int = x
      val r2 : int = y
      method getx = r1
      method gety = r2
      method eq (p: 'a) = (r1,r2) = (p#getx,p#gety)
end;;
# class ColorPoint (x,y,s) =
    object (self: 'b)
      val s1 : string = s
      method gets = s1
       inherit point (x,y)
      method eq (p1: 'b) =
      (r1,r2,s1) = (p1#getx,p1#gety,p1#gets)
end;;
# let m1 = new ColorPoint (1,2,"red") and n1 = new ColorPoint (1,2,"red");;
val m1 : ColorPoint = ⟨obj⟩
val n1 : ColorPoint = ⟨obj⟩
# m1# eq n1;;
- : bool = true
# let i = new Point (1,2) and j = new Point (3,4);;
val i : Point = ⟨obj⟩
val j : Point = ⟨obj⟩

# let npt = new Point (3,4);;
val npt : Point = ⟨obj⟩
# let breakit ( pt: Point) = if pt#eq npt then "ok" else "no";;
val breakit : Point → string = ⟨fun⟩
# breakit j;;
- : string = "ok"
# breakit m1;;
This expression has type
ColorPoint =
```

⟨ eq : ColorPoint → bool; getx : int; gety : int; gets : string ⟩
but is here used with type
Point = ⟨ eq : Point → bool; getx : int; gety : int ⟩
Only the first object type has a method gets
#

i# eq m1;;
This expression has type
Point = ⟨ eq : Point → bool; getx : int; gety : int ⟩
but is here used with type
ColorPoint =
⟨ eq : ColorPoint → bool; getx : int; gety : int; gets : string ⟩
Only the second object type has a method gets
■

The second solution is by using multi-methods, which are collection of methods with same name and different type signatures. When a multi-method is called, the selection of the method code depends on the classes of one or more of the parameters of the method, *not* just on the receiver type. This is referred as multiple dispatch. Using multi-methods, subclasses can be safely considered as subtypes. The solution to multi-methods is adapted in `Java`.

Example 7.2.2 import java.lang.*;

```
class Point
{ public int x=0;
     public int y=0;
     public Point(int a, int b)
    { this.x = a; this.y =b; }
     public int getx() { return(x);}
     public int gety(){ return(y);}

     public boolean eq(Point p)
    { System.out.println("pc "+this.x+" pc "+this.y);
     System.out.println("pc "+p.x+" pc "+p.y);
     return ( (this.x==p.x) & (this.y==p.y));}
     public void disp()
    { System.out.println("pc "+this.x+" pc "+this.y);};
     public boolean breakit()
     { Point nupt = new Point(1,2);
     return(this.eq(nupt)); }
}

class ColorPoint extends Point
    { public String s;
     public String gets() return(s);}
     public ColorPoint(int a, int b, String s1)
    { super(a,b);
     // this.x = a; this.y =b;
     this.s= s1; }
     public boolean eq(ColorPoint p)
    { System.out.println("cpc "+p.x+" cpc "+p.y+" cpc "+p.s);
     System.out.println("cpc "+this.x+" cpc "+this.y+" cpc "+this.s);
     return ((this.x==p.x) & (this.y==p.y) & (s == p.s));}
     public void disp()
```

```
    { System.out.println("cpc "+this.x+" cpc "+this.y+" cpc "+this.s);};
}

class Jbinari {
     public static void main (String args[]) {
     Point pt = new Point(1,3);
     ColorPoint cpnt1,cpnt2;
     cpt= new ColorPoint(1,2,"red");

            System.out.println("********** : 1 ");
            System.out.println(pt.breakit());

            System.out.println("********** : 2");
            System.out.println(cpt.breakit());
        }
}
```

■

Example 7.2.2 shows the classes *Point* and *ColorPoint*. There method *eq* is a multi-method formed by methods of the two classes *Point* and *ColorPoint*. The type of *eq* is

$$\{Point \times Point \to bool,\ ColorPoint \times ColorPoint \to bool\}$$

When *eq* is called by a pair of parameters, then the system will select the "best match" method for those parameters, *i.e.* if at least one of the parameters of the method *eq* is of type *Point* and the other is a subtype of it, then the method *eq* of class *Point* is selected for execution; if both parameters are of type a subtype *i.e. ColorPoint* then the method *eq* of class *ColorPoint* is executed. See Example 7.2.2, when the method *breakit* is called by *cp* of *ColorPoint*, *i.e.* *cp.breakit*(); then it selects the method *eq* of class *Point* for execution, since the second paramenter is of type *Point*. In general, when a multi-method m of type $\{X_{10} \times \ldots \times X_{1n} \to T_{11}, \ldots, X_{k0} \times \ldots \times X_{kn} \to T_{kn}$ is called, *i.e.* $X_0.m(X_1, X_2, \ldots, X_n)$, then the system selects the code in X at runtime for execution, where X is superclass of any X_i, *i.e.* $X_i \leq X,\ \forall\ i = 0..n$, and moreover, for any other X' superclass of all X_i'^{s}, $X \leq X'$. Since the selection of the method code is choosen at runtime, we have *dynamic dispatch*.

Finally, we point out that, in the realm of non-class-based OO-languages, analogous problems arise in the presence of binary methods, when we try to combine object extension with subtyping, see *e.g.* [14, 54]. In particular the solution discussed in [54] is probably related to the one presented in the next section for the class-based OO-language Fickle.

7.3 Yet Another Solution

In this section, we present yet another solution for typing binary methods in presence of inheritance. In the conclusion we will comment briefly on how this solution came about and how to possibly strengthen it. In our solution, arrow types, *i.e.* the types of methods, are contravariant, and a subclass is a subtype only when the types of overloaded methods are subtypes of the corresponding methods in the superclass. Under this view, $ColorPoint \not\leq Point$, since the type $ColorPoint \to bool$ of the method *eq* in the class *ColorPoint* is not a subtype of the type $Point \to bool$ of the overloaded method in the class *Point*. Thus, the method call *breakit(p,cp)*, where p is a *Point* and *cp* is a *ColorPoint*, is not permitted. However, the system as described above would be quite restrictive, since *e.g.* the call *breakit(p,cp)* would be forbidden, even if this does not cause any problem, as well as any call to a method with an actual parameter of type *ColorPoint*, in place of a formal parameter of type *Point*. Intuitively, only those method calls are problematic, where the specialized *ColorPoint* parameter is used, inside the method, as receiver of a binary overloaded method. Thus, if we are able to prove that a method m has a certain type, under the assumption that a given parameter belongs to the class *Point* without the method *eq*, then such parameter

can be safely substituted by a *ColorPoint* actual parameter. In order to formalize this intuition we extend the collection of types with types of the shape $C^{-m_1,\ldots,m_k}$, where C is the class and $m_1,\ldots,m_k$ are methods in C, and $C^{-m_1,\ldots,m_k}$ is meant to represent the class C without the methods $m_1,\ldots,m_k$.

Now, the method *breakit* can receive the type *bool breakit*($Point\ x_1, Point^{-eq}\ x_2$), since the parameter x_2 is not used as the target of an *eq* method call. Moreover, since $ColorPoint \leq Point^{-eq}$ (*eq* is not overloaded anymore) the method call *breakit(p,cp)*, for *p* of class *Point* and *cp* of class *ColorPoint*, is now allowed. In particular, all method calls with *ColorPoint* parameter in place of a *Point* parameter are permitted, when the method body does not contain any call to *eq*. Notice that, however, since the parameter x_1 is used as receiver of an *eq* method call, then breakit *cannot* be typed with *bool breakit*($Point^{-eq}\ x_1, Point\ x_2$), and thus the call *breakit(cp,p)* is correctly *not* permitted.

In order to exemplify our proposal, we implement it in the case of the programming language Fickle introduced in Chapter 2, whose coalgebraic semantics has been studied in Chapter 6. For simplicity, we will focus on the core of Fickle without reclassification. Thus, in particular all effects ϕ disappear in the typing rules of [30] (see Chapter 2, Section 2.4, Tables 2.6,2.7). In the original typing system of [30], the problem of binary methods is solved simply by forbidding overloading, and by allowing only overriding in subclasses. We recall that methods in subclasses which override methods in the superclass must have parameters with the same types as in the overridden methods of the superclass. This is expressed by the rule for wellformed classes (see Table 2.7, Chapter 2). Thus, in Fickle the class *ColorPoint* with method *bool eq*($ColorPoint\ x_1$) is not definable. This restriction on the formation of subclasses is sufficient to guarantee the safety of the system, where subclasses are considered as subtypes (see Table 2.5), and the arrow type is contravariant (see the rule *meth* for the method call in Table 2.6). This solution is the one adopted in many statically typed object oriented languages, such as C^{++}, as seen in Subsection 7.2. But, in our view this is a rather simplistic solution, which prevents natural forms of overloading, such as the method *eq* of *ColorPoint*. In our solution as explained above, we do not impose limitations on overloading, but rather we provide a fine notion of subtyping which does not coincide with the notion of subclass. In the following, we show how the typing system of [30] presented in Chapter 2, has to be modified in order to implement our solution.

$Types: \quad \tau ::= bool \mid int \mid C^{-m_1,\ldots,m_k}$
where $C^{-m_1,\ldots,m_k}$ represents the class C without the methods $m_1,\ldots,m_k$; when the list $m_1,\ldots,m_k$ is empty, then we write simply C, as usual.

The rule for wellformed classes of Table 2.7 becomes now more liberal, allowing for more general forms of overloading (ignoring reclassification):

$$\frac{\begin{array}{l}\mathcal{C}(P,c) = \texttt{class c extends c'}\ \{\ldots\} \\ \forall f:\ \mathcal{FD}(P,c,f) = t_0 \implies P \vdash t_0\ \diamond_{ft} \text{ and } \mathcal{F}(P,c',f) = Udf \\ \forall m:\ \mathcal{MD}(P,c,m) = \tau\ m(\tau_1 x_1,\ldots,\tau_n x_n)\,\{e\} \implies \\ \quad P, \{x_1:\tau_1,\ldots,x_n:\tau_n, this:c\} \vdash e:\tau'' \\ \quad P \vdash \tau'' \leq \tau \\ \quad \mathcal{M}(P,c',m) = \mathcal{U}df \text{ or} \\ \qquad (\mathcal{M}(P,c',m) = \tau'\ m(\tau'_1 x_1,\ldots,\tau'_n x_n) \text{ and } \tau \leq \tau', \tau_i \leq \tau'_i\ \forall i)\end{array}}{P \vdash c\ \diamond}$$

The subtyping relation of Table 2.5 has to be modified as follows. Subclasses are not automatically subtypes: they are subtypes only when types of overloaded methods are in the subtyping relation. More precisely, we substitute the last rule of Table 2.5 with the following:

```
class d extends Object {
    bool breakit (Point p1, Point p2)
        { p1.eq(p2); }
}
```

Figure 7.5: Fickle code for breakit

```
class d extends Object {
    bool breakit (Point p1, Point^-eq p2)
        { p1.eq(p2); }
}
```

Figure 7.6: New version for breakit

$$\frac{\begin{array}{l} P \vdash c' \sqsubseteq c \\ \forall m: \ \mathcal{MD}(P, c', m) = \tau'\ m(\tau'_1 x_1, \ldots, \tau'_n x_n) \text{ and} \\ \qquad \mathcal{MD}(P, c^{-m_1,\ldots,m_k}, m) = \tau\ m(\tau_1 x_1, \ldots, \tau_n x_n) \implies \\ \qquad \tau' \leq \tau \text{ and } \tau'_i \geq \tau_i \ \forall i\ ) \end{array}}{c' \leq c^{-m_1,\ldots,m_k}}$$

Now, in this modified system, the judgment $P, \{p_1 : ColorPoint, p_2 : Point\} \vdash breakit(p_1, p_2) : bool$ cannot be derived, since $P \nvdash ColorPoint \leq Point$. However, the judgment $P, \{p_1 : Point, p_2 : ColorPoint\} \vdash breakit(p_1, p_2) : bool$ is derivable, if we assign the type $Point^{-eq}$ to the second parameter of method breakit, as in the code of Figure 7.6. Namely, the class d in Figure 7.6 is wellformed, since $P, \{p_1 : Point, p_2 : Point^{-eq}\} \vdash p_1.eq(p_2)$ is still derivable using the typing rules for expressions of Table 2.6. Moreover, since $P \vdash ColorPoint \leq Point^{-eq}$ is derivable, then also $P, \{p_1 : Point, p_2 : ColorPoint\} \vdash breakit(p_1, p_2) : bool$ is derivable.

A result analogous to Theorem 2.4.1 holds for our new typing system, *i.e.* one can prove that our system is sound, in the sense that

Theorem 7.3.1 *For a well-formed program P, environment Γ, and expression e, such that*

$$P, \Gamma \vdash e \ : \ t$$

if $P, \Gamma \vdash \sigma\diamond$, and e, σ converges then

- $e,\ \sigma \ \longrightarrow_P \ v,\ \sigma', \quad P,\ \sigma' \vdash v \lhd\ t, \quad P, \Gamma' \vdash \sigma'\diamond,$

  or

- $e,\ \sigma \ \longrightarrow_P \ nullpntrExc,\ \sigma'.$

The solution for typing binary methods in presence of inheritance that we have presented in this section improves the original typing system for Fickle, by capturing more class definitions. Moreover it supports "future code extension" without loosing the subtyping property. Our solution is quite simple, in the end, but still based on single dispatching. However, it is not so general as the multi-method solution, since *e.g.* the call $breakit(cp_1, cp_2)$, where cp_1 and cp_2 are of type *ColorPoint*, is not accepted by our typing system, because method *breakit* cannot be typed by $bool(breakit\ Point^{-eq}\ x_1, Point^{-eq}\ x_2)$. In principle, we could cover this method calls still living in the world of single dispatching, at the price of making our system more complex.

Conclusions

In this thesis, we have extended the original coalgebraic model of Reichel-Jacobs for classes and objects of OO-languages in two ways: by including (generalized) binary methods, and by modelling also a language with the store. Moreover, we have addressed another critical topic of OO-languages, *i.e.* typing binary methods when subclasses are viewed as subtypes. While the coalgebraic analysis of generalized binary methods presented in Chapter 5 of this thesis appears rather satisfactory, still there are interesting issues which deserve further study. Here we list some of them:

- A coalgebraic account of infinitary methods *i.e.* methods whose parameters include an infinite sequence of sets or objects, appears to be very problematic. Is a coalgebraic account of these really hopeless?
- The freezing and the graph approach do not always coincide. We feel that when this happens the class is underspecified. Can this be formalized more stringently?
- To apply the coalgebraic approach to Featherweight Java.
- To explore coalgebraic accounts of component-based and service oriented computing.
- Case studies of applications of *coinduction principles* and *bialgebraic specification* in proving program correctness: we will build a suitable module for the interactive proof editor *COQ*;
- In Chapter 6, a coalgebraic model of an OO-language with the store is presented, but only unary methods are considered. Extra issues arise when binary methods are included, and the problem of providing a coalgebraic model in the general case remains open.
- The issue of program equivalence and program transformation for `Java`-like OO-languages seems not to have been investigated in the literature. In Chapter 2, we initiate such an investigation for Fickle, and in Chapter 6, we compare the bisimilarity equivalence induced by our coalgebraic model with the notion of observational equivalence introduced in Chapter 2. However, a more systematic study of notions of observational equivalences for `Java`-like OO-languages and a comparison with coalgebraic equivalence is called for.
- The graph coalgebraic approach of Chapter 5 for modelling binary methods was the original suggestion for the solution to the problem of typing binary methods when subclasses are considered as subtypes, presented in Chapter 7, Section 7.3.

  In fact, in the graph approach, methods are modelled as graphs instead of as functions, importing this idea in the world of types, we can introduce a new type constructor, *i.e.* the *relation type*, and use this to type binary methods in class declarations. Since relation types are purely covariant, the subtyping property is maintained by subclasses. Binary methods can still be typed also with the standard arrow type, which is a subtype of the corresponding relation type, see Table C.1. To preserve safety, contrary to arrow types, we assume relation

$$\frac{x : \alpha \quad M : \beta}{\lambda x^{\alpha}.M : \alpha \otimes \beta} \qquad \frac{\alpha \leq \alpha' \quad \beta \leq \beta'}{\alpha \otimes \beta \leq \alpha' \otimes \beta'} \qquad \frac{}{\alpha \to \beta \leq \alpha \otimes \beta}$$

Table C.1: Typing rules for *Relational Types* $\otimes$

types not to be "applicable", *i.e.* one can decide that there is no relational counterpart to the rule : $\frac{M:\alpha\to\beta \quad N:\alpha}{MN:\beta}$, this is the origin of the "reduced" types for the classes, or partial functions. This solution to the problem of typing binary methods is quite simple, in principle, and it allows for single dispatching in method calls. The syntax presented in Section 7.3 of Chapter 7 was introduced later in order to make the system more palatable.

The relation type should deserve a move through investigation for its own sake. We feel that multiple dispatch can be construed as a relational type, but more work needs to be done.

Appendix

Categorical Definitions

In this appendix, we collect some basic categorical definitions.

Definition 7.0.2 (Category) *A category $\mathcal{C}$ is given by the data (1)-(5), which satisfy the properties (i)-(iv):*

1. *a collection $Obj(\mathcal{C})$ of objects $\mathcal{C}$;*
2. *a collection $Mor(\mathcal{C})$ of morphisms(arrows, maps)of $\mathcal{C}$;*
3. *an operation, which assigns to each morphism $f \in Mor(\mathcal{C})$ objects $dom(f)$, $cod(f)$, called* domain and codomain *of f, respectively;*
4. *an operation, which assigns to each object $X \in Obj(\mathcal{C})$ a morphism id_X, called* identity morphism *on X;*
5. *an operation, which assigns to each pair of morphisms f, g such that $cod(f) = dom(g)$, a morphism $g \circ f$, called their* composition*(such a pair f, g is said to be composable) satisfying the following properties:*

   (i) $dom(id_X) = X = cod(id_X)$;

   (ii) $dom(g \circ f) = dom(f), \quad cod(g \circ f) = cod(g)$;

   (iii) for each $f \in \mathcal{C}$ with $X = dom(f)$ and $Y = cod(f)$ $f \circ id_X = f = id_Y \circ f$;

   (iv) if f, g, and g, h are composable pairs, then $h \circ (g \circ f) = (h \circ g) \circ f$.

Definition 7.0.3 (Functor) *Let $\mathcal{C}, \mathcal{D}$ be two categories. A* functor *$F : \mathcal{C} \to \mathcal{D}$ assigns to each object $X \in Obj(\mathcal{C})$ an object $F(X) \in Obj(\mathcal{D})$, and to each morphism $f \in Mor(\mathcal{C})$ a morphism $F(f) \in Mor(\mathcal{D})$, such that:*

- *if $f : X \to Y$ in $\mathcal{C}$, then $F(f) : F(X) \to F(Y)$ in $\mathcal{D}$;*
- $F(id_X) = F(id_{F(X)})$;
- $F(g \circ f) = F(g) \circ F(f)$.

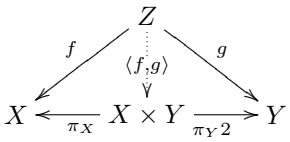
The notion of functor captures the notion of morphism between categories, while the notion of natural transformation (below) captures morphisms between functors.

Definition 7.0.4 *Given functors $F, G : \mathcal{C} \to \mathcal{D}$, a natural transformation $\lambda : F \dot{\Rightarrow} G$ is an operation which assigns to each object X in $\mathcal{C}$ an arrow $\lambda_X : F(X) \to G(X)$ in $\mathcal{D}$ such that, given any $f : X \to Y$ in $\mathcal{C}$, $Gf \circ \lambda_X = \lambda_Y \circ Ff$.*

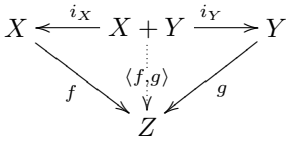
$$\begin{array}{ccc} F(X) & \xrightarrow{Ff} & F(Y) \\ {\scriptstyle \lambda_X}\downarrow & & \downarrow{\scriptstyle \lambda_Y} \\ G(X) & \xrightarrow[Gf]{} & G(Y) \end{array}$$

Throughout this appendix we work in a category $\mathcal{C}$. The objects of $\mathcal{C}$ are ranged over by X, Y, Z, *etc.*, while the morphisms are ranged over by e, f, g, *etc.*

Definition 7.0.5 (Product) *A* binary product *of X and Y is a triple $X \xleftarrow{\pi_X} X \times Y \xrightarrow{\pi_Y} Y$ such that, for all $f : Z \to X,\ g : Z \to Y$, there exists a unique $\langle f, g\rangle : Z \to X \times Y$ making the following diagram commute:*



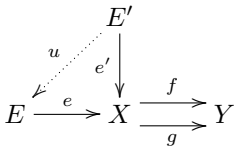
Definition 7.0.6 (Coproduct) *A* binary coproduct *of X and Y is a triple $X \xrightarrow{i_X} X + Y \xleftarrow{i_Y} Y$ such that, for all $f : X \to Z,\ g : Y \to Z$, there exists a unique $\langle f, g\rangle : X + Y \to Z$ making the following diagram commute:*



Definition 7.0.7 (Equalizer) *Let $f, g : X \to Y$. An* equalizer *of f and g is a pair $(E, e : E \to X)$ such that*

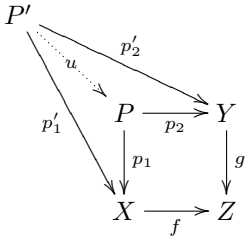
1. *$f \circ e = g \circ e$*, i.e. $E \xrightarrow{e} X \underset{g}{\overset{f}{\rightrightarrows}} Y$

2. *for any other equalizer $(E', e' : E' \to X)$, there exists a unique morphism $u : E' \to E$ such that the following diagram commute:*



Definition 7.0.8 (Pullback, Kernel Pair) *Let $f : X \to Z$ and $g : Y \to Z$.*

- *A* weak pullback *of f and g is a triple $(P, p_1 : P \to X, p_2 : P \to Y)$, such that, for any triple $(P', p_1' : P' \to X, p_2' : P' \to Y)$, there exists a morphism $u : P' \to P$ such that the following diagram commute:*



- *A* pullback *of f and g is a weak pullback such that the morphism u in the diagram above is* unique.

- *A* kernel pair *is a pullback for a morphism $f : X \to Y$ with itself.*

Definition 7.0.9 (Forgetful Functors) *Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A forgetful functor $U : \mathcal{C} \to \mathcal{D}$ is a functor which operates on objects of $\mathcal{C}$ by "forgetting" any imposed structure. Examples of forgetful functors are:*

- $U^L : Alg_L \to \mathcal{C}$, *mapping algebras to their carriers:*

$$(X, \beta) \overset{U^L}{\mapsto} X$$

- $U_H : Coalg_H \to \mathcal{C}$, *mapping coalgebras to their carriers:*

$$(X, \alpha) \overset{U_H}{\mapsto} X$$

- $U^\lambda : \lambda\text{-}Bialg \to Coalg_H$, *forgetting the algebra structure of* λ*-bialgebras:*

$$(X, \beta, \alpha) \overset{U^\lambda}{\mapsto} (X, \alpha)$$

- $U_\lambda : \lambda\text{-}Bialg \to Alg_L$, *forgetting the coalgebra structure of* λ*-bialgebras:*

$$(X, \beta, \alpha) \overset{U_\lambda}{\mapsto} (X, \beta)$$

Following [59], we define adjunction as follows:

Definition 7.0.10 (Adjunction) *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be categories. An adjunction from* $\mathcal{C}$ *to* $\mathcal{D}$ *is a triple* $\langle F, G, \varphi \rangle$*, where* $F : \mathcal{C} \to \mathcal{D}$ *and* $G : \mathcal{D} \to \mathcal{C}$ *are two functors, and* φ *is a function which assigns to each pair of objects* $X \in \mathcal{C}$ *and* $Y \in \mathcal{D}$ *a bijection*

$$\varphi = \{\varphi_{X,Y} : \mathcal{D}(F(X), Y) \cong \mathcal{C}(X, G(Y))\}_{X,Y}$$

which is natural in X *and* Y*.*

Here naturality of the bijection φ means that for all morphisms $h : X' \to X$ in $\mathcal{C}$ and all morphisms $k : Y \to Y'$ in $\mathcal{D}$ the following diagrams commute:

$$\begin{array}{ccc} \mathcal{D}(F(X), Y) & \xrightarrow{\varphi_{X,Y}} & \mathcal{C}(X, G(Y)) \\ {\scriptstyle (Fh)^*}\downarrow & & \downarrow{\scriptstyle h^*} \\ \mathcal{D}(F(X'), Y) & \xrightarrow[\varphi_{X',Y}]{} & \mathcal{C}(X', G(Y)) \end{array}$$

$$\begin{array}{ccc} \mathcal{D}(F(X), Y) & \xrightarrow{\varphi_{X,Y}} & \mathcal{C}(X, G(Y)) \\ {\scriptstyle k_*}\downarrow & & \downarrow{\scriptstyle (Gk)_*} \\ \mathcal{D}(F(X), Y') & \xrightarrow[\varphi_{X,Y'}]{} & \mathcal{C}(X, G(Y')) \end{array}$$

k_* is short for $\mathcal{D}(F(X), k)$, the operation of composition with k, and $h^* = \mathcal{C}(h, G(Y))$.

An adjunction is a bijection which assigns to each arrow $f : FX \to Y$ an arrow $\varphi f = \mathrm{rad} f : X \to GY$, the *right adjunct of* f, in such a way that the naturality conditions of above diagrams,

$$\varphi(f \circ Fh) = \varphi f \circ h, \qquad \varphi(k \circ f) = Gk \circ \varphi f,$$

hold for all f and all arrows $h : X' \to X$ and $k : Y \to Y'$. It is equivalent to require that φ^{-1} be natural; *i.e.* that for every h, k and $g : X \to GY$ one has

$$\varphi^{-1}(gh) = \varphi^{-1} g \circ Fh, \qquad \varphi^{-1}(Gk \circ g) = k \circ \varphi^{-1} g.$$

For such an adjunction, F is called the *left adjoint* of G and G the *right adjoint* of F. It is denoted as

$$F \dashv G.$$

Theorem 7.0.11 (Unit and Counit of adjuction) *An adjunction $\langle F, G, \varphi\rangle : \mathcal{C} \to \mathcal{D}$ determines*

1. *A natural transformation $\eta : 1_X \to GF$ such that for each object X the arrow η_X is universal to G from X, while the right adjunct of each $f : FX \to Y$ is*

$$\varphi f = Gf \circ \eta_X : X \to GY \tag{7.0.1}$$

   η is called unit of the adjunction.

2. *A natural transformation $\varepsilon : FG \to 1_Y$ such that for each arrow ε_Y is universal to Y from F, while each $g : X \to GY$ has left adjunct*

$$\varphi^{-1} g = \varepsilon_Y \circ Fg : FX \to Y \tag{7.0.2}$$

   ε is called counit of the adjuction.

3. *The following composites are identities:*

$$G\varepsilon \circ \eta G \qquad\qquad \varepsilon F \circ F\eta. \tag{7.0.3}$$

Summarizing we have:

Theorem 7.0.12 *Each adjunction $\langle F, G, \varphi\rangle : \mathcal{C} \to \mathcal{D}$ is completely determined by the items in any one of the following list:*

1. *Functors F, G, and a natural transformation $\eta : 1_X \to GF$ such that each $\eta_X : 1_X \to GFX$ is universal to X from G. Then φ is defined by Equation 7.0.1.*

2. *The functor $G : \mathcal{D} \to \mathcal{C}$, and for each $X \in \mathcal{C}$, and object $F_0 X \in \mathcal{D}$, there exists a universal arrow $\eta_X : 1_X \to GF_0X$ from X to G. Then the functor F has object function F_0 and is defined on arrows $h : X \to X'$ by $GFh \circ \eta_X = \eta_X \circ h$.*

3. *Functors F, G, and a natural transformation $\varepsilon : FG \to 1_Y$ such that each $\varepsilon_Y : FGY \to 1_Y$ is universal to F from Y. Then φ^{-1} is defined by Equation 7.0.2.*

4. *The functor $F : \mathcal{C} \to \mathcal{D}$, and for each $Y \in \mathcal{D}$, and object $G_0 Y \in \mathcal{C}$, there exists a universal arrow $\varepsilon_Y : FG_0Y \to 1_Y$ from F to Y. Then the functor G has object function G_0 and is defined on arrows $k : Y' \to Y$ by $\varepsilon_Y \circ FGk = k \circ \varepsilon_{Y'}$.*

5. *Functors F, G, and a natural transformation $\eta : 1_X \to GF$ and $\varepsilon : FG \to 1_Y$ such that both composites in Equation 7.0.3 are the identity transformations. Here φ is defined by Equation 7.0.1 and φ^{-1} is defined by Equation 7.0.2.*

Bibliography

[1] M. Abadi, L. Cardelli. A Theory of Objects. *Monographs in Computer Science*, Springer, 1996.

[2] S. Abramsky, L. Ong. Full Abstraction in the Lazy Lambda Calculus, *Information and Computation* **105**(2), 1993, 159–267.

[3] P. Aczel. Non-wellfounded sets, *CSLI* Lecture Notes **14**, Stanford 1988.

[4] P. Aczel. Final Universes of Processes. In *MFPS'93*, Brookes et al. eds., LNCS **802**, 1993.

[5] P. Aczel. The initial algebra and final coalgebra perspectives, Four lectures given at the 1995 "Logic of Computation" Advanced Study Institute International Summer School at Marktoberdorf, in *Logic of Computation*, edited by H. Schwichtenberg, Springer 1997, 1–3.

[6] P. Aczel, N. Mendler. A Final Coalgebra Theorem. In *CTCS*, D.H.Pitt et al. eds., Springer LNCS **389**, 1989, 357–365.

[7] J. Adamek, S. Milius, J. Velebil. Final Coalgebras and a solution theorem for arbitrary endofunctors. In *ENTCS*, **65**(1), 2002.

[8] J. Adamek, S. Milius, J. Velebil. On Coalgebra based on Classes. In *TCS*, **316**, 2004, 3–23.

[9] J. Adamek, S. Milius, J. Velebil. A General Final Coalgebra Theorem. In *MSCS*, **15**, Cambridge University Press 2005, 409–432.

[10] P. America, J. de Bakker, J. N. Kok, J. Rutten. Denotational semantics of a parallel object-oriented language. In *Information and computation*, **83**(2), 1989, 152 – 205.

[11] F. Bartels. On generalised coinduction and probabilistic specification formats. PhD thesis, CWI Amsterdam, 2004.

[12] J. Beck. Distributive Laws. *Seminar on Triples and Categorical Homology*, B. Eckmann, editor, Springer-Verlag **80**, 1969, 119–140.

[13] J. van den Berg, M. Huisman, B. Jacobs, E. Poll. A type-theoretic memory model for verification of sequential Java programs, *WADT'99*, D. Bert, C. Choppy, editors, Springer LNCS **1827**, 1999, 1–21.

[14] V. Bono, L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In Proc. of *CSL*, International Conference of Computer Science Logic, Springer LNCS, **933**, 1995, 16-30.

[15] K. B. Bruce, L. Cardelli, G. Castagna, J. Eifrig, S. F. Smith, V. Trifonov, G. T. Leavens, B. C. Pierce On Binary Methods, *TAPOS* **1**(3), 1995, 221-242.

[16] R. Bruni, F. Honsell, M. Lenisa, M. Miculan. Modeling fresh names in the pi-calculus using abstractions, in *CMCS'04* Conference Proceedings, J. Adamek ed., ENTCS vol. 106, 2004, 25-41.

[17] D. Cancila, F. Honsell, M. Lenisa. Some Properties and Some Problems on Set Functors, in *CMCS'06*, ENTCS, to appear.

[18] G. Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems* **17**(3), 1995, 431–447.

[19] G. Castagna. Object-Oriented Programming: A Unified Foundation. Progress in *TCS*, Birkauser, Boston, 1997.

[20] G. Castagna, G. Ghelli, G. Longo. A calculus for overloaded functions with subtyping. In *Information and computation*, **117**(1), February 1995, 115-135.

[21] L. Cardelli, P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys* 17(**4**), 1986, 471–522.

[22] C. Chambers. Predicate Classes. *ECOOP'93*, Springer LNCS **707**, 1993, 268–296.

[23] C. Cirstea. Integrating observations and computations in the specification of state-based, dynamical systems, PhD thesis, University of Oxford, 2000.

[24] W. R. Cook. *Object-Oriented Programming Versus Abstract Data Types.* In the Proc. of the *REX Workshop/School on the Foundations of Object-Oriented Languages*, Springer LNCS **173**, 1990, pp. 151-178.

[25] W. R. Cook, W. L. Hill, P. S. Canning. Inheritance is not subtyping. 17th Ann. In *ACM* Symp. on Principles of Programming Languages, (ACM Sigplan, 1990), pp. 125-135.

[26] A. Corradini, R. Heckel, U. Montanari. Tile Transition Systems as Structured Coalgebras. In FCT'99, LNCS **1684**, 1999, pp. 13- 38.

[27] A. Corradini, R. Heckel, U. Montanari. Compositional SOS and beyond: A coalgebraic view of open systems, *TCS* **280**, 2002, 163–192.

[28] S. Drossopoulou. Three Case Studies in $Fickle_{II}$, Tech. rep., Imperial College, `http://www.di.unito.it/~damiani/papers/dor.html`.

[29] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Giannini. Dynamic Object Reclassification: $Fickle_{II}$. In *ECOOP'01*, J. L. Knudsen et al. eds, Springer LNCS **2072**(2), 2001, 130–149.

[30] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Giannini. More Dynamic Object Re-classification: $Fickle_{II}$. In *ACM Transactions On Programming Languages and Systems* **24**(2), 2002, 153–191.

[31] H. Ehrig, B. Mahr. Fundamentals of Algebraic Specifications 1 : Equations and Initial Semantics, *ENTCS Monographs on TCS*, W. Brauer et al. eds., Springer-Verlag, 1985.

[32] M. Forti, F. Honsell. Set-theory with free construction principles, *Ann. Scuola Norm. Sup. Pisa*, Cl. Sci.**10** (4), 1983, 493–522.

[33] G. Ghelli and D. Palmerini. Foundations of Extended Objets with Roles (extended abstract). In *FOOL6*, 1999.

[34] J. Goguen, G. Malcolm. A Hidden Agenda. In *TCS* **245**, 2000, 55-101.

[35] J. Goguen, J. Thatcher, E. Wagner, J. Wright. Initial algebra semantics and continuous algebras. In *ACM* **24**(1), 1977, 68–95.

[36] J. Guttag. The Specification and Application of Programming of Abstract Data Types. PhD thesis, Univ. of Toronto, 1975.

[37] J. Guttag. Abstract data types and the development of data structures. *Communications of the ACM* **20**(6), 1977, 297–404.

[38] R. Hennicker, A. Kurz. (Ω, Ξ)-Logic: On the algebraic extension of coalgebraic specifications. In *CMCS'99*, ENTCS **19**, 1999.

[39] U. Hensel, M. Huisman, B. Jacobs, H. Tews. Reasoning about Classes in Object-Oriented Languages: Logical Models and Tools, *European Symposium on Programming*, C. Hankin, editor, Springer LNCS **1381**, 1998, 105–121.

[40] C. Hermida, B. Jacobs. Structural induction and coinduction in a fibrational setting. In *Information and Computation* **145**(2), 1998, 107-152.

[41] F. Honsell, M. Lenisa. Final Semantics for Untyped Lambda Calculus. In *TLCA'95*, G. Plotkin,M. Dezani editors, Springer LNCS **902**, Berlin 1995, 249-265.

[42] F. Honsell, M. Lenisa. Coinductive Characterizations of Applicative Structures, *Math. Struct. in Comp. Science* **9**, 1999.

[43] F. Honsell, M. Lenisa, U.Montanari, M.Pistore. Final Semantics for the π-calculus, *PROCOMET'98*, D. Gries et al. eds, Chapman & Hall, 1998.

[44] F. Honsell, M. Lenisa, R. Redamalla. Coalgebraic Semantics and Obsevational Equivalances of an Imperative Class-based OO-Language, *COMETA'03*, F. Honsell et al. eds., ENTCS **104**, 2004, 163-180.

[45] F. Honsell, M. Lenisa, R. Redamalla. Coalgebraic Description of Generalized Binary Methods, in the Proceedings of *DCM'05*, M. Fernandez, I. Mackie editors, ENTCS **135**(3) 2006, 73-84.

[46] M. Huisman. Reasoning about Java programs in Higher-order logic using PVS and Isabelle, Phd thesis, University of Nijmegen, The Netherlands.

[47] B. Jacobs. Objects and Classes, Co-algebraically, *Object-Orientation with Parallelism and Book Persistence*, B. Freitag et al. eds., Kluwer Academic Publishers, 1996, 83–103.

[48] B. Jacobs. Inheritance and cofree constructions. In *ECOOP'96*, P. Cointe, editor, Springer LNCS **1098**, 1996, 210–231.

[49] B. Jacobs. Behaviour-refinement of object-oriented specifications with coinductive correctness proofs. In *TAPSOFT'97*, M. Bidoit, M. Dauchet, editors, Springer, LNCS **1214**, 1997, 787–802.

[50] B. Jacobs, J. Rutten. A tutorial on (co)algebras and (co)induction. In *Bulletin of the EATCS* **62**, 1996, 222–259.

[51] M. Lenisa. Final Semantics for a Higher Order Concurrent Language. In *CAAP'96*, H. Kirchner et. al. eds., LNCS **1059**, 1996, 102–118.

[52] M. Lenisa, J. Power, H. Wantanabe. Distributive of endofunctors, pointed and co-pointed endofunctors, monads and comonads. In *CMCS'99* Conference Proceeding, B. Jacobs and J. Rutten editors, ENTCS **33**, 2000.

[53] X. Leroy. The Objective Caml System: Documentation and User s Manual, 2002, at www.ocaml.org.

[54] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In Proc. of *ECOOP*, Springer Verlag, LNCS **1241**,1997, 146-169.

[55] B. Liskov, A, Snyder, R. Atkinson, C. Schaffert. Abstraction mechanisms in CLU. In *Communications of the ACM* **20**, 1977, 564–576.

[56] B. Liskov, J. Wing. A behaviour notion of subtyping, *ACM Trans. on Prog. Lang. and Systems*, 1994, 1811–1841.

[57] B. Liskov, J. Wing. Behavioral subtyping using invariants and constraints. Technical Report CMU CS-99-156, School of Computer Science, Carnegie Mellon University, July 1999.

[58] B.H. Liskov, S.N. Zilles. Programming with Abstract Data Types. In *ACM SIG-PLAN Noties*, 1974, 50–59.

[59] S. Mac Lane. Categories for the Working Mathematician. Springer Verlag, 1971.

[60] B. Meyer. Object-oriented Software Construction. Prentice Hall, New York, N.Y., second edition, 1997.

[61] J. C. Mitchell, G. D. Poltkin. Abstract types have existential type. In the Proceedings of *ACM Symp. on Principles of Programming Languages,* 1985.

[62] C. Morgan. Programming from Specifications: Second Edition. Prentice Hall International, Hempstead, UK, 1994.

[63] E. Poll, J. Zwanenburg. From algebras and coalgebras to dialgebras. In *CMCS'01*, Corradini et al. eds., ENTCS **44**, 2001.

[64] H. Reichel. An approach to object semantics based on terminal co-algebras. In *MSCS* **5**, 1995, 129-152.

[65] D. Remy. Using, Understanding, and Unraveling the OCaml Language. in *Applied Semantics: Advanced Lectures*, G. Barthe, editor, LNCS **2395**, Springer Verlag 2002,413–537.

[66] D. Remy, J. Vouillon. Objective ML: A simple object oriented extension of ML. In *ACM Symposium on Principles of Programming Languages*, 1997, 40–53.

[67] G. Rosu, J. Goguen. Hidden Congruent Deduction. In *FTP'98*, LNAI **1761**, 2000, 252-267.

[68] J. Rothe, H. Tews, B. Jacobs. The Coalgebraic Class Specification Language CCSL. In the Journal of Universal Computer Science, 7(2001), pp.175–193.

[69] J. J. M. M. Rutten. Universal coalgebra: a theory of systems, *TCS* **249**(1), 2000, 3–80.

[70] J. J. M. M.Rutten, D. Turi. On Foundations of Final Semantics: Non-Standard sets, Metric Spaces, Partial Orders, J.de Bakker et al. eds., LNCS **666**, 1993, 477–530.

[71] J. J. M. M.Rutten, D. Turi. *REX*, J.de Bakker et al. eds., LNCS **803**, 1994, 530–582.

[72] J. J. M. M.Rutten, D. Turi. On the Foundations of coalgebra semantics: Non-Well founded sets, Partial Orders, Metric Spaces. *MSCS* **8**(5), 1998, 481–540.

[73] D.Sannella, A.Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* **9**, 1997, 229–269.

[74] M. Serrano. Wide Classes. In *ECOOP'99*, R. Guerraoui, editor, LNCS **1628**, Springer, 1999, 391–415.

[75] A. Tailvasaari. Object Oriented Programming with Modes. In the Journal of Object Oriented Programming, **6**(3), 1993, 27–32.

[76] H. Tews. Coalgebras for Binary Methods. In *CMCS'2000*, ENTCS **33**, 2000.

[77] H. Tews. Coalgebraic Methods for Object-Oriented Specifications, Ph.D. thesis, Dresden Univ. of technology, 2002.

[78] D. Turi, G. Plotkin. Towards a mathematical operational semantics 12^{th} *LICS*, IEEE, Computer Science Press, 1997, 280–291.

[79] D. Ungar, R. B. Smith. Self: The power of simplicity. *OOPSLA'87* Conference Proceedings, Orlando, FL, October, 1987, pp. 227-241. Published as *SIGPLAN Notices* **22**(12), December, 1987.

[80] M. Wirsing. Algebraic specification. *Handbook of TCS Vol* **B**, J. van Leeuwen et al. eds., Elsevier Science Publishers, 1990, 675–788.

[81] M. Wirsing. Algebraic specification Languages. In *Recent trends in Data Type Specification, 10th workshop on Specification of Abstract Data Types*,E. Astesiano, G Reggio, A. Tarlecki, editors, 1994, 81–115. Springer-Verlag, 1995.