1996

Algorithmic Abstraction Via Polymorphism In Object-oriented Programming Languages

Qingyu Zhuang

Algorithmic Abstraction via Polymorphism in Object-Oriented Programming Languages

by

Qingyu Zhuang

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
November 1995

ISBN 0-612-09893-1

ABSTRACT

The abstraction gap between algorithms and functions (procedures) causes numerous duplicate efforts in implementing the same algorithms as well as apparent under-use of many efficient algorithms. A solution to this problem is to realize algorithmic abstraction at the programming language level in addition to data abstraction. For this purpose, a programming language needs to have (1) an adequate type hierarchy for defining the domains of algorithms and (2) proper constructs for writing the sequences of steps of algorithms that are polymorphic to all types of objects in an algorithm domain.

Many object-oriented programming languages use a two-level hierarchy of entities (objects and classes) to support data abstraction. This hierarchy is only adequate to support two forms of polymorphism: universal-bounded polymorphism and inclusion-bounded polymorphism. Neither of them is proper for algorithmic abstraction.

The main aim of this thesis is to develop an adequate type structure and proper forms of polymorphism for algorithmic abstraction in object-oriented programming languages. We define a four-level hierarchy of entities: objects, classes, types, and kinds. Objects and classes have ordinary meanings; a type is a set of classes that have exactly the same external behaviour; and a kind is a set of types that share common properties. Based on the type hierarchy, we establish various forms of polymorphism which allow us to write polymorphic functions at different abstraction levels. We show that kinds are at the proper abstraction level for defining the domains of algorithms, and that kind-bounded polymorphic functions are algorithms which we propose to be supported at the programming language level.

Another goal of this thesis is to develop an object-oriented algebraic theory for the semantics of the four-level type hierarchy. In our approach, a kind corresponds to a structured signature which can have other signatures as its sorts. Thus, any algebra of a structured signature is a structured algebra which can have other algebras as its domains. We show that a structured algebra fits naturally into the structure of a class.

Keywords: object-oriented programming language, algorithmic abstraction, parametric polymorphism, kind-bounded quantification, structured signature, structured algebra

ACKNOWLEDGEMENTS

Just as many friends have said, I am always lucky in career and life! My supervisor, Professor Sheng Yu, has given me a "golden" project and allowed me to enjoy the happiness of success. Although my path was not clear at first, it is hard work and perseverance that have brought the meaning of success. From the bottom of my heart, thank you, Sheng. Without your invaluable advice, endless support, and generous encouragement over the past years, it is certainty that I would not reach this point.

Many thanks also go to Professor MingHua Zhang who visited the department from Oct. 1992 to March 1993. During his visit, he gave me much meaningful advice and guidance.

I would like to thank Professor Andy Szilard for his helpful comments on the object-oriented algebraic theory introduced in this thesis.

I would like to thank Mr. Peter G.N. Scheyen for his industrious work in implementing Kind-C++. Without his work, the author would not have concentrated on the theoretical work of Kind-C++.

I would also like to thank this department for the financial support. In particular, I wish to thank Ms. Sandy Alfs and Ms. Ursula Dutz for their help in the last couple of years.

Finally I wish to give my thanks to my wife, Peng Wang. She has held me while I cried with frustration, leapt for joy with me when things were going well, and, in general, provided me with the most important support I could have asked for. Special thanks would be given to my parents-in-law, who have given my family support during the past years.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Each year many new and efficient algorithms are published in various journals and conference proceedings. Algorithm designers (theoreticians) publish their algorithms in *pseudo*-code, which in practice need to be re-digested and re-coded into functions (or procedures) by programmers. Hence, from the programmers' perspective, it is desirable that algorithm designers write their algorithms in directly compilable code. Each algorithm is usually applicable to an infinite set of data types, and it is impossible for algorithm designers to write a compilable function for each applicable data type. Thus, programmers often use algorithms that are easy to understand and re-code but inefficient and out-of-date.

These phenomena reflect the well-known gaps between algorithms and functions, and between algorithm designers and practical programmers. By an *algorithm*, we mean "*a precise and unambiguous specification of a sequence of steps*" ([1]) that can be applied to a certain range of types of objects and always terminates. There are two major parts in an algorithm definition:

(1) a domain, i.e., a collection of data types to which the algorithm is applicable,

(2) a sequence of steps that specify a particular computational approach.

For example, the domain of the Quicksort algorithm consists of all vector types that have a total order, and the step sequence of the algorithm specifies a divide-and-conquer sorting paradigm. Since the domain of an algorithm usually includes a variety of types, each step of the step sequence cannot be type specific; for example, the use of the English word "compare" instead of the operator symbol "<". In contrast, a *function* is a sequence of statements written in a certain programming language. Hence, a function can be applied only to a specific type of objects.

An algorithm is essentially an abstraction of a group of functions (procedures). In other words, from an algorithm programmers can obtain concrete functions for proper types. Hence, an algorithm can be regarded as a function from a domain of types to concrete functions. However, an algorithm domain is at a higher abstraction level than a function domain. For example, a graph algorithm applies to all data types that possess the properties of a graph, while a function implementing the algorithm can apply only to a specific data type. Obviously, once an algorithm is coded as a polymorphic function (which is a function from a domain of types to concrete functions) in a programming language, we can obtain concrete functions simply by calling it with proper types. Hence, it is desirable to write algorithms as compilable polymorphic functions.

Object-oriented programming languages (OOPLs) have laid a foundation for algorithmic abstraction since they provide data abstraction and some forms of polymorphism which are necessary for algorithmic abstraction. However, the object-oriented concept itself does not encompass the idea of algorithmic abstraction. To support algorithmic abstraction properly, an object-oriented programming language has to be endowed with

(1) a type hierarchy to support the definition of algorithm domains, and

(2) constructs to define the step sequences of algorithms that are polymorphic to all types of objects in a given domain.

More precisely, for (1) above, the commonly used two-level type hierarchy consisting of objects and classes does not adequately support the definition of algorithm domains; at least one more level of entities is needed to express the properties that are shared by data types. For example, we should be able to specify all data types that possess the properties of a graph and all data types that have a total order. For (2) above, polymorphisms with universal quantification and bounded quantification are either too coarse or too restrictive [17]. There should be a more general form of polymorphism than the above two forms. The main aim of this thesis is to develop an appropriate type hierarchy and proper forms of polymorphism for algorithmic abstraction in OOPLs.

1.1 A Perspective on Related Issues

In this section, we will briefly review the two-level type hierarchy as well as existing forms of polymorphism to provide the point of departure for this thesis.

1.1.1 Two-Level Type Hierarchy

Many OOPLs are based on the two-level type hierarchy consisting of objects and classes. Such languages include C++ [53, 55], Smalltalk [31], and Eiffel [43]. An object is an integrated unit of data and methods, and a class is an implementation unit that consists of data structures together with methods manipulating the data structures. Objects with exactly the same methods and data structures are also said to form a class.

In the object-oriented programming environment, the domain of an algorithm is a set of classes that satisfy a set of properties imposed by the algorithm. For example, the domain of the Quicksort algorithm contains all vector classes that have a total order. Since classes are implementation-dependent entities, they are not appropriate for defining algorithm domains. Although some simple mechanisms, such as subclasses and class templates in C++ [53, 55], do exist in the two-level type hierarchy, they are not general enough for this purpose. For example, all classes that have a total order are not subclasses of the same class, and they also cannot be defined by a class template.

Abstract classes in C++ can be used to specify the properties that are shared by classes. Abstract classes are a special kind of classes. A class satisfies the properties defined by an abstract class if it is a subclass of the abstract class. Objects of a subclass can be treated as those of a superclass [53, pp.184]. Hence, if two classes are declared to be subclasses of an abstract class, then their objects can be used where objects of the abstract class are expected. However, this is not what we expect sometimes. Consider an abstract class C that is used to define all classes that have a total order. Then we derive two classes C_1 and C_2 from C. C_1 is for complex numbers, and C_2 is for strings. Since both objects of C_1 and objects of C_2 can be treated as objects of C, objects of C_1 would be compared with objects of C_2. However, it is meaningless to compare a complex number with a string. Hence, abstract classes cannot define the following restriction: the two parameters of an algorithm can be both of class C_1 or both of class C_2 but should not be of C_1 and C_2, respectively. Signatures in G++ [5] and types in POOL-1 [2] also suffer from the same problem as abstract classes in C++.

One objective of this thesis is to develop an appropriate type hierarchy for OOPLs in which algorithm domains can be properly defined. Besides objects and classes, we need entities which not only are at higher abstraction levels than classes but also allow us to define the properties shared by classes.

We first realize that classes and types are totally different although they are often used as interchangeable concepts in the two-level type hierarchy. That is, a type is a set of classes that have exactly the same external behaviour, and a class is a specific implementation of a type. For example, we can implement two integer-stack classes with a linked list and an array, respectively. Although they have different internal representations (which are hidden from the outside), they have exactly the same external behaviour from the users' point of view. Hence, they are two implementations of the integer-stack type. It is obvious that any algorithm applicable to one integer-stack class is also applicable to another integer-stack class. Hence, types should be introduced as independent entities to specify classes that have exactly the same external behaviour.

An algorithm is applicable to a set of classes that have common properties but do not necessarily have exactly the same external behaviours. For example, we can specify an algorithm that selects the maximum member from two objects of a class. The domain of the algorithm contains all classes that have a total order, such as classes implementing pairs of integers and classes implementing pairs of strings. Hence, we also need to introduce entities to specify the properties that are shared by different types of classes.

1.1.2 Polymorphism

Polymorphism, first introduced by Strachey [52], allows us to write programs that can be applied to different data types. Without polymorphism, an algorithm has to be implemented for each type to which it is applied.

Polymorphic (second-order) λ-calculus (which was independently invented by Girard [27] and Reynolds [49]) has played a central role in the development of polymorphic typing systems [17]. However, polymorphic λ-calculus is a mathematical model that is well suited to functional programming languages. Hence, it has been used mainly for functional programming languages. For example, the polymorphic typing system of ML can be defined in terms of polymorphic λ-calculus [44, 45].

Cardelli and Wegner are the pioneers who applied polymorphic λ-calculus to OOPLs [14, 17]. They extended polymorphic λ-calculus with records and record types, where a record models an object and a record type models a type of objects. Their calculus supports two forms of polymorphism which are realized by ***universal quantification*** and ***bounded quantification***, respectively. Later, ***F-bounded quantification*** was introduced by Mitchell *et al.* to type functions over recursive types [13]. However, as we will argue below, they are not sufficiently expressive for algorithmic abstraction.

In universal quantification, a polymorphic function is expressed as a function that has a type variable ranging over the set of all types. Its use can be illustrated by the following function (In this thesis, we define functions using λ-expressions [18, 21] rather than a specific programming language):

$$\textbf{Fun}\ precede = (\forall t).\lambda x, y : t.\ \underline{if\ x \leq y\ then\ true\ else\ false}$$
$$precede : (\forall t).t \times t \rightarrow bool$$

where t is a type variable with universal quantification, x and y are two arguments restricted to t, and the function body is underlined. By the definition of universal quantification [17], type variable t can be replaced by any type. Hence, *precede* would be suitable for comparing two elements of any type. However, the operator "$\leq$" is used in the body of *precede*, and as a result *precede* can only be applied to such types that have "$\leq$" as an operator, such as *int* and *real* (We will only consider the object-oriented version of each type). Thus, universal quantification is a coarse way of specifying type variables. In many cases, universal quantification cannot prevent improper types from being used as actual types of a polymorphic function. In other words, if *precede* is applied to two objects of some type that does not have "$\leq$" as an operator, an unpredictable evaluation will result. To avoid such abnormal phenomena, users have to read through the entire body of the function to determine what operators each valid substituting type for a type variable should possess. This is just what we try to avoid [38]. Notice that function templates in C++ are an example of universal quantification [53]. The problems associated with universal quantification will be addressed from a practical point of view later.

In bounded quantification, a polymorphic function is expressed as a function that has a type variable restricted to the set of all subtypes of a given type. An object of a subtype can be treated as one of its supertype according to the subtyping principle [14]. Hence, a polymorphic function in bounded quantification would work properly on all subtypes of a given type. Let T be a type that has the comparison operator "$\leq$". In bounded quantification, *precede* can be rewritten for all subtypes of T as follows:

$$\textbf{Fun}\ precede' = (\forall t \leq T).\lambda x, y : t.\ \underline{if\ x \leq y\ then\ true\ else\ false}$$
$$precede' : (\forall t \leq T).t \times t \rightarrow bool$$

where type variable t is restricted to all subtypes of T using "$\forall t \leq T$". Obviously, bounded quantification is different from universal quantification since it can specify a type variable that ranges over the set of all subtypes of a given type rather than the set of all types. Users of such a polymorphic function need only read its heading to determine what types can be used as actual types rather than reading through its entire body each time. Note that *int* and *string* are two proper types of function *precede* since they both have the operator "$\leq$". Hence, T should be such a type that has both *int* and *string* as its subtypes. Suppose that such a supertype is defined for *int* and *string*. Then, by the subtyping principle, both *int* objects and *string* objects can be used where objects of the supertype are expected. For instance, an *int* object would be compared with a *string* object. But such a comparison is undefined and does not make a sense. Therefore, we do not have a common supertype for both *int* and *string*. Thus, bounded quantification is too restrictive in this case.

Bounded quantification also does not work properly when recursive types are considered. A few examples can be found in [13]. Thus, F-bounded quantification, an extension of bounded quantification, was introduced to type functions over recursive types [13]. However, we can find functions where F-bounded quantification does not work. Consider the type function $list(t) = \{v : T,\ link : t\}$, where T is still the type that has the operator "$\leq$". It is obvious that "$\leq$" can be used to sort a list of objects of type T. In F-bounded quantification, we can write a sorting function which has type $(\forall t \leq list(t)).t \rightarrow t$. According to the definition of F-bounded quantification, a type T' is a proper type of the sorting function if it satisfies the condition $T' \leq list(T')$ (which means that T' must be a subtype of type $\{v : T,\ link : T'\}$). Let $T_1 = \{v : int,\ link : T_1\}$ and $T_2 = \{v : string,\ link : T_2\}$. T_1 is a type for lists of integers, and T_2 is a type for lists of strings. As we have shown, there does not exist a supertype for both *int* and *string*. Hence, it does not hold that *int* is a subtype of T and *string* is a subtype of T. As a result, T_1 is not a subtype of $list(T_1)$ or T_2 is not a subtype of $list(T_2)$. Thus, the sorting function is not applicable to T_1 or T_2 although its function body may be used to sort any object that belongs to T_1 or T_2. Therefore, the sorting function cannot work in the way we might expect.

1.2 The Goal

The goal of this thesis is to develop an appropriate type hierarchy and proper forms of polymorphism for algorithmic abstraction in OOPLs. In order to meet the requirement of algorithmic abstraction, we introduce a four-level hierarchy of entities consisting of *objects*, *classes*, *types*, and *kinds*. Based on this hierarchy, we can define polymorphic functions at various abstraction levels. In particular, kind-bounded polymorphic functions represent algorithms which we propose to be supported at the programming language level.

In the four-level type hierarchy, objects and classes have ordinary meanings. A type is a set of classes that have exactly the same external behaviour, and a kind is a set of types that share a set of common properties. For example, we can define a type for integer stacks which can have any valid implementation, such as arrays or linked lists. Similarly, we can define a type for complex-number stacks. Integer stacks and complex-number stacks are of different types since integers and complex-numbers are of different types. However, they both satisfy the "*LIFO*" property. Hence, we can define a kind which contains both integer stacks and complex-number stacks.

Note that kinds cannot be replaced by supertypes. Furthermore, an arbitrary pair of distinct types in a kind may have neither of the following relations: (1) one is a subtype of the other; (2) both of them are subtypes of another type in the kind. Details of the four-level hierarchy are presented in Chapter 3.

All classes of a type have exactly the same methods from the external point of view. Hence, a function defined over a class may be generalized to a polymorphic function that is applicable to all the classes that belong to the same type as the class. Such a polymorphic function (called a *type-bounded* function) is defined using *type-bounded* quantification, which introduces a class variable restricted to a type. Similarly, all types of a kind share a common set of properties. Hence, if a function is defined over some type of a given kind and uses only the operators that are specified by the kind, then it can be generalized to a polymorphic function that is applicable to all types of the kind. Such a polymorphic function (called a *kind-bounded* function) is defined using *kind-bounded* quantification, which introduces a type variable restricted to a kind.

To some extent, a type-bounded function can be regarded as a language-supported algorithm since it can be used as a function from classes defined by a type to concrete functions. However, all classes of a type have to have exactly the same external behaviour. As a result, types are too restrictive when they are used to define algorithm domains. In contrast, a kind defines a set of types that have common properties and is specified by these properties. In addition, an algorithm often demands only certain properties on each type in its domain. As well, two or more parameters of an algorithm may be of different types that satisfy the same set of properties. All of these characteristics can be properly defined by kinds. Therefore, kinds are well suited for defining algorithm domains.

In essence, a kind-bounded function is a function from types defined by a kind to concrete functions. Hence, kind-bounded functions represent language-supported algorithms which have kinds as their domains. Different from algorithms in the form of *pseudo*-code, language-supported algorithms are specified using precise programming languages. Therefore, they can be used directly without any translation.

This thesis is organized as follows.

In Chapter 2, the basic concepts of polymorphism and λ-calculus are introduced. The reader familiar with these concepts may skip to Chapter 3.

In Chapter 3, we define a four-level hierarchy of entities: objects, classes, types, and kinds. We generalize the concept of inheritance so that it can support the incremental development of types and kinds.

In Chapter 4, we deal with various forms of polymorphism that can be established in the four-level hierarchy of entities. We show that polymorphism with kind-bounded quantification can be used to specify language-supported algorithms.

In Chapter 5, a language called Kind-C++ (an extension of C++) is presented to illustrate the use of algorithmic abstraction.

In Chapter 6, we develop an object-oriented algebraic theory to define the semantics of the four-level type hierarchy. In our approach, a class is modelled as a structured algebra, a type as a set of isomorphic structured algebras, and a kind as a set of structured algebras that have common properties.

In Chapter 7, we compare our approach with other related works. We give our concluding remarks for this thesis and offer some suggestions for future research.

Chapter 2

Polymorphism and λ-Calculus

In this chapter, we give a brief introduction to the concepts that are needed to understand this thesis. Readers familiar with these concepts can omit this chapter.

2.1 Polymorphism

Many programming languages such as *Pascal* and *Fortran* only allow functions and procedures to be applied to values of a unique type. Such languages have the following disadvantage: if an algorithm is needed for a number of types, then different source code must be created for them. For example, if a *bubble sort* algorithm is needed for arrays of *integer* and *string*, respectively, in *Pascal*, two different subroutines have to be defined. It is a tedious work for programmers to repeat almost the same coding. Such programming languages are said to be *monomorphic* [17].

In contrast, *polymorphic* programming languages allow functions and procedures to be applied to operands of more than one type. For example, in C++ ([53]) the bubble sort algorithm can be implemented with a function template that is applicable to any type of vectors that has the operator "<". Obviously, polymorphism provides the facility for programmers to write reusable programs.

Strachey distinguished two kinds of polymorphism: *ad-hoc polymorphism* and *parametric polymorphism* [52]. Cardelli and Wegner refined Strachey's classification by introducing *inclusion polymorphism*, which is used to model subtype and inheritance in OOPLs [17].

2.1.1 Ad-hoc Polymorphism

In *ad-hoc* polymorphism, a function can work in different ways on a range of types. We can distinguish two kinds of *ad-hoc* polymorphism: *overloading* and *coercion*.

By overloading, we mean that the same identifier is used to denote different functions and the context is used to decide which function is being used for a particular computation. Consider the following expressions:

(1) $30 + 100$,

(2) $23.2 + 12.1$,

(3) "*abcd*" + "*efghij*".

Here, "+" denotes the addition over *int*, the addition over *real*, and the concatenation over *string*, respectively. Thus, "+" represents different operators in different type contexts. Obviously, overloading makes programs easy to write.

In coercion, a value is converted to one of an expected type. Coercion can be done statically at compile-time or dynamically at run-time. Suppose that *real* has precedence over *int* in the situation where "+" needs two operands of the same type. The following expressions will be converted as they are expected:

(1) $12.1 + 10 \rightarrow 12.1 + 10.0$,

(2) $12 + 10.2 \rightarrow 12.0 + 10.2$.

In OOPLs, objects of a subtype need to be coerced into those of its supertype where objects of the supertype are expected. Consider the following two record types.

$aa = \{a : string,\ b : int\}$

$bb = \{a : string,\ b : int,\ c : string\}$

Any object of type *bb* can be coerced into one of type *aa* by ignoring its field *c*. Hence, type *bb* is a subtype of type *aa*. Thus, objects of type *bb* can be used where objects of type *aa* are expected. Henceforth, if it is not stated explicitly, we will consider coercion only in the object-oriented environment.

2.1.2 Parametric Polymorphism

In parametric polymorphism, a function is represented as one that has a type variable restricted to a certain range of types. A type variable can be replaced by actual types. The functions that implement parametric polymorphism are called ***function templates*** in C++ [53], ***generic functions*** in *Ada* [3], and ***polymorphic functions*** in *ML* [56]. A few methods of implementing parametric polymorphism are discussed in [47].

Parametric polymorphism supports the reuse of functions at the programming language level. That is, a parameterized polymorphic function can be evoked uniformly with values of different types. For example, a C++ function template for the bubble sort algorithm can be called with various types of vectors [53, pp. 271]. Thus, parametric polymorphism greatly reduces the size of source code.

We can distinguish two kinds of parametric polymorphism, i.e., ***true*** parametric polymorphism and ***apparent*** parametric polymorphism [17]. True parametric polymorphism also supports the reuse of functions at the machine code level; i.e., the same piece of machine code for a function can be applied to various types of values. The uniformity of machine code requires that all data to a parameterized polymorphic function be represented uniformly. Obviously, such a piece of machine code may not be optimal for a specific type. In contrast, apparent parametric polymorphism does not support the reuse of functions at the machine code level. Instead, a function in apparent parametric polymorphism is only a ***macro*** that will be instantiated with actual types at compile-time. Thus, different machine code will be generated for different types.

2.1.3 Inclusion Polymorphism

In inclusion polymorphism, a function is represented as one that has a type variable restricted to the set of all subtypes of a given type. Clearly, inclusion polymorphism is a special form of parametric polymorphism. Consider the following type for persons.

Type $P = \{name : string, age : int\}$

Intuitively, a type is a subtype of type P if it includes all fields of type P. For example, type S for students includes all fields of type P plus the field $class : string$. Hence S is a subtype of P. Now we define a polymorphic function as follows:

Fun $add\_age = (\forall t \leq P).\lambda x \in t.\ \underline{x.age + 1}$

where type variable t is restricted to all subtypes of P. Hence, it can be applied to objects of both P and S.

2.2 λ-Calculus

λ-calculus has been developed to study the general properties of functions in mathematics [18, 21, 48]. In λ-calculus, functions are represented with symbolic notations called λ-expressions. Since λ-expressions are mathematical formulae, we can manipulate them directly and evaluate their values by performing expression transformations.

The influence of λ-calculus on programming languages is quite obvious. For example, the design of the programming language *Lisp* has been largely based on λ-calculus [42, 25], and the *call by name* mechanism used in many languages is closely related to the substitution operation defined in λ-calculus.

Girard and Reynolds independently developed polymorphic λ-calculus [28, 49]. Since then, polymorphic λ-calculus has been used as one of the most important tools to deal with polymorphism in programming languages. Several semantic models of polymorphic λ-calculus were developed by Bruce *et al.* [9, 10, 46]. To deal with inheritance and subtyping in OOPLs, polymorphic λ-calculus was extended with records and record types by Cardelli *et al.* [15, 14, 17]. The semantics of the extended polymorphic λ-calculus was established by Bruce and Longo [8].

The development of programming languages from an untyped structure to polymorphism can be reflected by the development of λ-calculus. In the rest of this section, we give a brief introduction to untyped λ-calculus, typed λ-calculus, and polymorphic λ-calculus.

2.2.1 Untyped λ-Calculus

The untyped λ-calculus is not concerned with the domain and the range of a function; instead, it focuses only on the properties that are common to all functions. Expressions in untyped λ-calculus have the following syntax [48]:

$$
\begin{aligned}
<\lambda\text{-}expression> &::= <variable> \mid <constant> \mid <application> \mid \\
&\quad\ <abstraction> \\
<application> &::= (<\lambda\text{-}expression>) <\lambda\text{-}expression> \\
<abstraction> &::= \lambda <variable> . <\lambda\text{-}expression>
\end{aligned}
$$

Variables and constants are the simplest λ-expressions, and more complicated λ-expressions can be constructed from them using *application* and *abstraction*, which are the only two constructing operations in untyped λ-calculus.

An *application* is simply the application of one λ-expression to another λ-expression; the former is called an operator, and the latter is called an operand. Obviously, a λ-expression can be used as an operator or an operand without any restriction at all.

An *abstraction* is used to form a function in the following way: the expression following "." is the function body. Of course, we can form functions with more than one argument by repeatedly using abstractions. The following are two λ-expressions that are formed by using the syntax specified above:

$$(1)\ \lambda x.x. \qquad\qquad (2)\ \lambda f.\lambda x.(f)x.$$

The first expression defines the *identity* function that returns x for any argument x. In the second expression, f is a function variable that corresponds to a function parameter in programming languages.

Definition 2.1 An occurrence of a variable x in a λ-expression e is said to be *bound* if it is in a part of e with the form $\lambda x.e''$; otherwise it is *free*. If x has at least one free occurrence in e, then it is called a free variable of e.

If two λ-expressions differ only in the names of their bound variables, then they are considered to be the same. For example, both $\lambda f.\lambda x.(f)x$ and $\lambda g.\lambda y.(g)y$ represent the same function although they use different names for their arguments. In fact, x and f in $\lambda f.\lambda x.(f)x$ correspond to formal parameters of functions in programming languages where arbitrary names can be chosen for formal parameters.

Definition 2.2 The *renaming* of a variable x to a variable y in a λ-expression e, denoted $[y/x]e$, is defined inductively as follows:

(1) $[y/x]x = y$,

(2) $[y/x]z = z$, if $x \neq z$,

(3) $[y/x]\lambda x.e = \lambda y.[y/x]e$,

(4) $[y/x]\lambda z.e = \lambda z.[y/x]e$, if $x \neq z$,

(5) $[y/x](e_1)e_2 = ([y/x]e_1)[y/x]e_2$.

The definition also states that we can change the names of variables in a λ-expression. The fact can be expressed in the following rule:

<u>α-conversion</u>: $\lambda x.e \Rightarrow_\alpha \lambda y.[y/x]e$, for any y that does not occur in e.

The expression on the left-hand side of "$\Rightarrow_\alpha$" is said to be *α-convertible* to the one on the right-hand side of "$\Rightarrow_\alpha$". Obviously, if e_1 is α-convertible to e_2, then e_2 is also α-convertible to e_1. Hence, α-convertibility is a symmetric relation over λ-expressions.

Definition 2.3 The *substitution* of e' for all free occurrences of variable x in e, denoted $\{e'/x\}e$, is defined by induction as follows:

(1) $\{e'/x\}x = e'$,

(2) $\{e'/x\}y = y$, if $x \neq y$,

(3) $\{e'/x\}\lambda x.e = \lambda x.e$,

(4) $\{e'/x\}\lambda y.e = \lambda y.\{e'/x\}e$ if $x \neq y$, and x is not free in e or y is not free in e',

(5) $\{e'/x\}\lambda y.e = \lambda z.\{e'/x\}[z/y]e$ if $x \neq y$, x is free in e, and y is free in e', where z is a variable which is neither free nor bound in $(e)e'$.

(6) $\{e'/x\}(e_1)e_2 = (\{e'/x\}e_1)\{e'/x\}e_2$.

β-reduction: $(\lambda x.e)e' \Rightarrow_\beta \{e'/x\}e$.

The β-reduction is used to evaluate λ-expressions. The following is an example that shows how a λ-expression is evaluated:

$$\begin{aligned}
& (\lambda x.\lambda y.(x)y)\lambda u.(y)u \\
= \; & \{\lambda u.(y)u/x\}\lambda y.(x)y \\
= \; & \lambda z.\{\lambda u.(y)u/x\}[z/y](x)y \\
= \; & \lambda z.\{\lambda u.(y)u/x\}([z/y]x)[z/y]y \\
= \; & \lambda z.\{\lambda u.(y)u/x\}(x)z \\
= \; & \lambda z.(\{\lambda u.(y)u/x\}x)\{\lambda u.(y)u/x\}z \\
= \; & \lambda z.(\lambda u.(y)u)z.
\end{aligned}$$

The evaluation of a λ-expression is terminated only when no more subexpressions can be simplified. Such a λ-expression is said to be in ***normal form***. There exist λ-expressions for which the reduction never terminates. However, if a λ-expression has a normal form, the normal form is unique up to α-convertibility. This property is called the ***diamond property*** [19, 35, 36].

2.2.2 Typed λ-Calculus

The typed λ-calculus is like the untyped λ-calculus except that each λ-expression has a type. Its syntax is defined as follows:

$$\begin{aligned}
&<\lambda\text{-}expression> ::= <variable> \mid <constant> \mid <application> \mid \\
&\qquad\qquad\qquad\qquad <abstraction> \\
&<application> \quad ::= (<\lambda\text{-}expression>) <\lambda\text{-}expression> \\
&<abstraction> \quad ::= \lambda <variable>:<type> . <\lambda\text{-}expression> \\
&<type> \quad ::= <ground\ type> \mid <function\ type> \\
&<function\ type> ::= <type> \rightarrow <type> \\
&<ground\ type> \quad ::= <basic\ type> \mid <structured\ type>
\end{aligned}$$

Types *int* and *real* are examples of basic types, and *list* and *record* are examples of structured types. Each variable is restricted to a type by a *type assignment*, which is a function from variables to types.

Let A be a type assignment, x be a value variable, and τ be a type expression. Then $A \cup \{x : \tau\}$ is the type assignment which is identical to A except that $(A \cup \{x : \tau\})(x) = \tau$. We define $A \models e : \tau$, which is read "e has type τ under A", by the following derivation system:

(1) If x is a variable and $A(x) = \tau$, then $A \models x : \tau$.

(2) If $A \cup \{x : \sigma\} \models e : \tau$, then $A \models \lambda x : \sigma. e : \sigma \rightarrow \tau$.

(3) If $A \models e : \sigma \rightarrow \tau$ and $A \models e' : \sigma$, then $A \models (e)e' : \tau$.

A typed λ-expression is evaluated if and only if a well-formed type can be inferred from it. Type-checking in typed λ-calculus prevents illegal λ-expressions from being used as operands of a given λ-expression. This is essentially the same as type-checking in programming languages.

2.2.3 Polymorphic λ-Calculus

Polymorphic λ-calculus is an extension of typed λ-calculus in that it allows type variables to be present in λ-expressions. Thus, if t is a type variable and e is a λ-expression, then $\Lambda t.e$ is also a λ-expression. A value variable can be restricted to a type or a type variable.

Definition 2.4 Let t denote type variables. Type expressions are defined as follows:

(1) Basic types are type expressions;

(2) Type variables are type expressions;

(3) If σ, τ are type expressions, then $\sigma \rightarrow \tau$ is a type expression;

(4) If τ is a type expression, then $(\forall t).\tau$ is a type expression.

An occurrence of a type variable t in a type expression τ is said to be *bound* if it is in a part of τ with the form $(\forall t).\sigma$; otherwise it is *free*. If t has at least one free occurrence in τ, then it is called a free type variable of τ. A type variable t is said to be *free* in a type assignment A if there exists a variable x such that t is free in $A(x)$.

Definition 2.5 Let x denote value variables, and t denote type variables. Then polymorphic λ-expressions can be defined inductively as follows:

(1) x is a polymorphic λ-expression;

(2) If e and e' are polymorphic λ-expressions, then $(e)e'$ is a polymorphic λ-expression;

(3) If τ is a type expression and e is a polymorphic λ-expression, then $\lambda x:\tau.\,e$ is a polymorphic λ-expression;

(4) If e is a polymorphic λ-expression, then $\Lambda t.e$ is a polymorphic λ-expression;

(5) If e is a polymorphic λ-expression and τ is a type expression, then $(e)\tau$ is a polymorphic λ-expression.

Let A be a type assignment, and e be a polymorphic λ-expression. We define $A \models e:\tau$, which means that e has type τ under A, by the derivation system below:

(1) If $A(x) = \tau$, then $A \models x:\tau$;

(2) If $A \models e:\sigma \rightarrow \tau$ and $A \models e':\sigma$, then $A \models (e)e':\tau$;

(3) If $A \cup \{x:\tau\} \models e:\sigma$, then $A \models \lambda x:\tau.\,e:\tau \rightarrow \sigma$;

(4) If $A \models e:\tau$, then $A \models \Lambda t.e:(\forall t).\tau$, where t is not free in A;

(5) If $A \models e:(\forall t).\tau'$, then $A \models (e)\sigma:\{\sigma/t\}\tau'$;

where $\{\sigma/t\}\tau'$ denotes the type expression that is obtained from τ' by replacing all free occurrences of t with σ. Note that it is not guaranteed that each λ-expression e is typable with respect A.

2.2.4 Remarks

λ-expressions are mainly used to represent functions and their applications. Let f and a be two λ-expressions. If f is interpreted as a function and a is interpreted as an argument of f, then $(f)a$ is interpreted as the result of applying f to argument a. Although $(f)a$ is a commonly used notation in λ-calculus, the more practical notation for function application is $f(a)$. In this thesis, if no ambiguity is caused in the evaluation of a λ-expression, we will use $f(a)$ to denote the application of function f to argument a.

In essence, $\Lambda t.e$ corresponds to a polymorphic function in programming languages, where t is an unrestricted type variable. To emphasize that t is a type variable ranging over the set of all types, we will use $(\forall t).e$ rather than $\Lambda t.e$ to represent such a polymorphic function. Similarly, $f(T)$ denotes the application of polymorphic function f to type T.

Chapter 3

A Four-Level Hierarchy of Entities

In this chapter, we formally define the four-level hierarchy of entities: ***objects***, ***classes***, ***types***, and ***kinds***. As we will see, each entity at a given level (except the object level) defines a set of entities at the next lower level. Hence, entities at a higher level are also at a higher abstraction level. In the following, we first introduce the hierarchy, and then we address various relations that exist among these entities. We show that the four-level type hierarchy is an adequate type structure for defining algorithm domains. The following notations will be used throughout this chapter:

ϑ (probably with a subscript) - denotes an object;

ξ (probably with a subscript) - denotes a class;

τ, σ (probably with a subscript) - denotes a type;

κ (probably with a subscript) - denotes a kind.

3.1 Objects, Classes, Types, and Kinds

3.1.1 Objects

An *object* is an integrated unit of data and methods. Each object has an identifier, *data* fields, and *method* fields. The data fields store the internal state of the object, while the method fields are functions that modify the internal state of the object and interface the object with the outside world.

In essence, a data field is a variable that carries a datum of a certain type. A data field can be accessed from the outside by defining trivial methods that simply return or update the value of the data field. Hence, we assume that all data fields of an object are hidden from the rest of a program. Not all methods of an object can be accessed from the outside of the object. We call those methods of an object that can be directly accessed from the outside the *properties* of the object.

An object does not necessarily have private methods. For simplicity, we assume that all methods are public. Thus, an object has two portions: the *private* portion and the *public* portion. The private portion of an object consists of data, and the public portion of an object consists of methods. Formally, an object can be defined using the following syntax:

$$\vartheta ::= < \{a_1 = e_1, \cdots, a_m = e_m\}, \{b_1 = e'_1, \cdots, b_n = e'_n\} >$$

where a_i and b_j come from a finite set of labels, and e_i and e'_j are expressions, $1 \leq i \leq m$, $1 \leq j \leq n$. $\{a_1 = e_1, \cdots, a_m = e_m\}$ is the private portion of ϑ, where a_i is a variable and e_i is an object of a certain type, $1 \leq i \leq m$. $\{b_1 = e'_1, \cdots, b_n = e'_n\}$ is the public portion of ϑ, where b_j is an identifier and e'_j is a function, $1 \leq j \leq n$. For the purpose of explanation, we restrict the kinds of expressions that can be used to define functions in this chapter. The syntax of expressions we use is given below:

$$e ::= \vartheta^{\xi} \mid x \mid self \mid \lambda x : \eta.\, e \mid e.a \mid update(e_1, a, e_2) \mid if\ e\ then\ e_1\ else\ e_2$$

where ϑ^{ξ} denotes an object of class ξ, and x denotes an object variable. The keyword "*self*" is a special object variable which denotes the object being defined. $\lambda x : \eta.\, e$ is a function in which x is an argument restricted to *method type* η (which is defined later) and e is the function body. $update(e_1, a, e_2)$ is a primitive function that updates the field a of e_1 with e_2 as the new value. The following is an object for the person named *John Li*:

```
Object JohnLi = < { Name = "John Li", Age = 25 },
                  { name   = {self.Name},
                    age    = {self.Age},
                    aging  = λn : int. {update(self, Age, self.Age + n)},
```

$$younger = \lambda p : \textbf{thisclass}.\ \{\ if\ (self.Age \leq p.age())\ then\ true\ else\ false\ \}$$

$$\} >$$

where *self* means object *JohnLi*, and **thisclass** is a special class variable that denotes the class *JohnLi* belongs to.

3.1.2 Classes

All objects that have exactly the same methods and data structures for their internal states form a *class*. Classes are implementation-dependent entities. For example, a stack class with an array implementation and a stack class with a linked-list implementation are considered to be different classes. Different objects of the same class have different identifiers and different locations in memory. In practice, it is not necessary for each object to keep its own methods. Instead, only one copy of each method is stored, and objects access their methods through pointers.

3.1.2.1 Class Definitions

At the programming language level, a class consists of a ***data scheme***, a ***method scheme***, and an *implementation* for its methods. Only the method scheme is open to the outside. The data scheme of a class defines the names and types of variables that carry the internal states of the objects of the class. The method scheme of a class defines the names and input/output types of its public methods (***properties***). The methods of a class are defined in its implementation section. A class is said to be a *basic* class if its method scheme does not involve other classes.

A class can be modelled as a 3-tuple which consists of a data scheme, a method scheme, and an implementation. For the purpose of presentation, we use the following syntax for classes:

$$\xi ::= < \{a_1 : \eta_1, \cdots, a_m : \eta_m\}, \{b_1 = e'_1 : \eta'_1, \cdots, b_n = e'_n : \eta'_n\} >$$

where a_i is a variable of *method type* η_i, $1 \leq i \leq m$, and b_j is a method that has expression e'_j as its method body and η'_j as its method type, $1 \leq j \leq n$. Thus, $\{a_1 : \eta_1, \cdots, a_m : \eta_m\}$, $\{b_1 : \eta'_1, \cdots, b_n : \eta'_n\}$, and $\{b_1 = e'_1, \cdots, b_n = e'_n\}$ corresponds to the data scheme, method scheme, and implementation of ξ, respectively. For simplicity, we will often express ξ as $< \{a_i : \eta_i\}_{1 \leq i \leq m}, \{b_j = e'_j : \eta'_j\}_{1 \leq j \leq n} >$. The syntax of method types is given below:

$$\eta ::= c \mid \xi \mid \textbf{thisclass} \mid \eta_1 \hookrightarrow \eta_2 \mid \tau$$

where c denotes a class variable, and ξ denotes a class. The keyword **thisclass** is a special class variable that denotes the class being defined, $\eta_1 \hookrightarrow \eta_2$ is the method type of methods from η_1 to η_2, and τ is a type expression. Class ξ is said to be a *parameterized* class if there exists at least one class variable in the definition of ξ. A parameterized class will be denoted by $\xi(\bar{c})$, where $\bar{c}$ is a list of class variables that occur in ξ.

Any method of a class will perform a certain computation on the internal states of objects of the class. However, these internal states are not visible to the outside. Hence, in practice, we do not explicitly provide internal states as actual parameters when we call methods. For example, if s denotes an object of an integer-stack class, then $s.push(10)$ will change the internal state of s by pushing "10", where the current state of s does not appear as an actual parameter of method $push$. For any class ξ, we use (ξ) to denote the set of internal states of its objects, and call (ξ) the internal type of ξ. There are three kinds of methods:

(1) *altering methods*: they alter the internal states of objects, but do not return any value.

(2) *outputting methods*: they do not change the internal states of objects, but return values related to them.

(3) *alter-outputting methods*: they not only change the internal states of objects, but also return values related to them.

An alter-outputting method can be realized using altering methods and outputting methods. Hence, we assume that only altering methods and outputting methods can be used in the definition of a class. Since the internal states of objects are always used implicitly as actual parameters of altering methods and outputting methods, we will not explicitly declare (ξ) to be an input type of a method of class ξ. But (ξ) can be used to declare an output type of a method of ξ, which means that the internal state of an object will be changed after the method is applied to the object. Obviously, internal types can only appear in the method types of altering methods.

Example 3.1 The following is a class for persons. In the class definition, we use the object variable *self*. Whenever an object of class *person* is created, *self* in the object will denote the object itself.

$$
\begin{aligned}
\textbf{Class}\ person = <\ \{\ Name : string,\ Age : int\ \},\qquad&\\
\{\ name \quad &= \{self.Name\}\\
&: \ \hookrightarrow string,\\
age \quad &= \{self.Age\}\\
&: \ \hookrightarrow int,\\
aging \quad &= \lambda n : int.\{update(self, Age, self.Age + n)\}\\
&: int \hookrightarrow (\textbf{thisclass}),\\
younger &= \lambda p : \textbf{thisclass}.\{\ if\ (self.Age \leq p.age())\\
&\qquad\qquad\qquad\qquad then\ true\ else\ false\ \}\\
&: \textbf{thisclass} \hookrightarrow bool\\
\}\ >&
\end{aligned}
$$

In the above example, **thisclass** denotes class *person*, and therefore (**thisclass**) denotes the internal type of *person*. As we will see, whenever *person* is inherited by other classes, **thisclass** has to be changed to indicate new classes.

3.1.2.2 Class Checking

Now we define a class-checking system that can be used to statically check the method type of an expression. The class-checking system is based on a function, called a

class assignment, **which associates each object variable with a class. Let A be a class assignment. Then $A' = A \cup \{x : \eta\}$ is the same as A except that $A'(x) = \eta$.**

We use $\frac{A}{B}$ to denote the fact that B is true if A is true. An expression e is said to be of method type η under A, denoted $A \models e : \eta$, if $e : \eta$ can be derived from A using the following rules.

$$A \models \vartheta^{\xi} : \xi \tag{3.1}$$

$$A \cup \{x : \xi\} \models x : \xi \tag{3.2}$$

$$A \models \mathit{self} : \textbf{thisclass} \tag{3.3}$$

$$\frac{A \cup \{x : \eta\} \models e : \eta'}{A \models \lambda x : \eta .\, e : \eta \hookrightarrow \eta'} \tag{3.4}$$

$$\frac{A \models e_1 : \eta_1,\ \cdots,\ A \models e_m : \eta_m,\ A \models e'_1 : \eta'_1,\ \cdots,\ A \models e'_n : \eta'_n}{A \models < \{a_i = e_i\}_{1 \le i \le m}, \{b_j = e'_j\}_{1 \le j \le n} > : < \{a_i : \eta_i\}_{1 \le i \le m}, \{b_j = e'_j : \eta'_j\}_{1 \le j \le n} >} \tag{3.5}$$

$$\frac{A \models e : < \{a_i : \eta_i\}_{1 \le i \le m}, \{b_j = e'_j : \eta'_j\}_{1 \le j \le n} >}{A \models e.a_i : \eta_i \quad (1 \le i \le m)} \tag{3.6}$$

$$\frac{A \models e : < \{a_i : \eta_i\}_{1 \le i \le m}, \{b_j = e'_j : \eta'_j\}_{1 \le j \le n} >}{A \models e.b_j : \eta'_j \quad (1 \le j \le n)} \tag{3.7}$$

$$\frac{A \models e_1 : \xi,\ A \models e_1.a : \eta,\ A \models e_2 : \eta}{A \models \mathit{update}(e_1, a, e_2) : (\xi)} \tag{3.8}$$

$$\frac{A \models e : \eta_1 \hookrightarrow \eta_2,\ A \models e' : \eta_1}{A \models e(e') : \eta_2} \tag{3.9}$$

$$\frac{A \models e : \mathit{bool},\ A \models e_1 : \eta,\ A \models e_2 : \eta}{A \models \mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \eta} \tag{3.10}$$

In essence, the above rules define an algorithm for class-checking. We have the following program scheme for class-checking, which is based on a partial function $\mathcal{T}$: Exp $\rightarrow$ Env $\rightarrow$ MethodType. Exp and MethodType denote expressions and

method types, respectively. **Env = Var → MethodType** denotes class assignments for object variables. The program returns a method type for an expression, or fails in the case of class errors. When $\mathcal{T}\|e\|\nu = \eta$, we mean that e is an expression of method type η under class assignment ν. When $\mathcal{T}\|e\|\nu = \mathit{fail}$, we mean that the class-checking of e fails.

$$\mathcal{T}\|x\|\nu =_{def} \nu(x)$$

$$\mathcal{T}\|\vartheta^{\xi}\|\nu =_{def} \xi$$

$$\mathcal{T}\|\mathit{self}\|\nu =_{def} \textbf{thisclass}$$

$$\mathcal{T}\|\lambda x : \eta .\, e\|\nu =_{def} \eta \hookrightarrow \mathcal{T}\|e\|(\nu \cup \{x : \eta\})$$

$$\mathcal{T}\| < \{a_1 = e_1, \cdots, a_n = e_m\}, \{b_1 = e'_1, \cdots, b_n = e'_n\} > \|\nu =_{def}$$
$$< \{a_1 : \mathcal{T}\|e_1\|\nu, \cdots, a_m : \mathcal{T}\|e_m\|\nu\}, \{b_1 = e'_1 : \mathcal{T}\|e'_1\|\nu, \cdots, b_n = e'_n : \mathcal{T}\|e'_n\|\nu\} >$$

$$\mathcal{T}\|e.b\|\nu =_{def} \text{if } \mathcal{T}\|e\|\nu =< \{\cdots\}, \{\cdots, b = e' : \eta', \cdots\} > \text{ then } \eta' \text{ else } \mathit{fail}$$

$$\mathcal{T}\|e.a\|\nu =_{def} \text{if } \mathcal{T}\|e\|\nu =< \{\cdots a : \eta \cdots\}, \{\cdots\} > \text{ then } \eta \text{ else } \mathit{fail}$$

$$\mathcal{T}\|\mathit{update}(e, a, e')\|\nu =_{def} \text{if } \mathcal{T}\|e.a\|\nu = \mathcal{T}\|e'\|\nu \text{ then } (\mathcal{T}\|e\|\nu) \text{ else } \mathit{fail}$$

$$\mathcal{T}\|e(e')\|\nu =_{def} \text{if } \mathcal{T}\|e\|\nu = \eta \hookrightarrow \eta' \text{ and } \mathcal{T}\|e'\|\nu = \eta \text{ then } \eta' \text{ else } \mathit{fail}$$

$$\mathcal{T}\|\mathit{if}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2\|\nu =_{def} \text{if } \mathcal{T}\|e\| = \mathit{bool} \text{ and } \mathcal{T}\|e_1\|\nu = \mathcal{T}\|e_2\|\nu$$
$$\text{then } \mathcal{T}\|e_1\|\nu \text{ or } \mathcal{T}\|e_2\|\nu \text{ else } \mathit{fail}$$

3.1.3 Types

A *type* is a collection of classes that have exactly the same external behavior, i.e., equivalent properties, but possibly different implementations. In particular, all basic classes with exactly the same external behaviour form a *basic* type. By two corresponding properties of two classes being equivalent, we mean that they are defined on exactly the same classes (types) except their internal types, and any two corresponding sequences of applications of their equivalent properties from their respective initial internal states should end with exactly the same external results and corresponding internal states under an isomorphism. A more precise definition can be found in Chapter 6.

Consider two integer-stack classes S_1 and S_2 with implementations by a linked list and an array, respectively. The stack properties are $push_1(x)$, $pop_1()$, $top_1()$, and $empty_1()$ for S_1 and $push_2(x)$, $pop_2()$, $top_2()$, and $empty_2()$ for S_2. Obviously, $push_1 : int \rightarrow (S_1)$ and $push_2 : int \rightarrow (S_2)$, ..., $empty_1 : \rightarrow boolean$ and $empty_2 : \rightarrow boolean$, are all of the same types, respectively, except for their internal types. In addition, if any two corresponding sequences of applications of stack properties on S_1 and S_2 from their respective initial states, e.g.,

$$push_i(X_1);\ push_i(X_2);\ pop_i();top_i();empty_i();pop_i();$$

for $i = 1,2$ and $X_1, X_2 \in int$, produce the same result, then S_1 and S_2 are of the same type. The type for all integer-stack classes can be defined as follows:

$$\textbf{type}\ stack = \{\ push : int \rightarrow (\textbf{thistype}),\ pop : \rightarrow (\textbf{thistype}),$$
$$top : \rightarrow int,\ empty : \rightarrow bool\ \}$$

where **thistype** denotes *stack*, and therefore (**thistype**) denotes the internal type of *stack*. Since type *stack* is independent of any class, there is no data structure in its definition. Each property is also called an ***abstract method***, which is independent of any implementation. For simplicity, the behaviour of a property is defined by an identifier and understood by programmers' intuition and common sense [62].

3.1.3.1 Type Definitions

According to the way a type is formed, we can distinguish two kinds of types: ***basic types*** and ***structured types***. A structured type is constructed from basic types and other structured types. The following is the syntax of types we use:

$$\tau ::= T \mid t \mid \textbf{thistype} \mid \{a_1 : \tau_1, a_2 : \tau_2, \cdots, a_n : \tau_n\} \mid \tau_1 \rightarrow \tau_2$$

where T denotes a basic type, and t denotes a type variable. The keyword **thistype** is a special type variable which denotes the type being defined.

Basic Types. Types *nat*, *bool*, and *char* are typical examples of basic types. Each basic type has a set of operators associated with it. A basic type may be represented in different ways. Hence, it is possible that a basic type corresponds to a number of implementations. In the terminology of object-oriented programming, a basic type may be implemented by different classes. See [6] for examples where basic types are defined with C++ classes.

Function Types. A *function* type from type τ_1 to type τ_2 is expressed as $\tau_1 \rightarrow \tau_2$, which is the type of all functions from τ_1 to τ_2. For example, the function *succ* over *nat* has the function type $nat \rightarrow nat$.

Record Types. A *record* type is an unordered set of labelled types. For example, $\{a : int,\ b : real\}$ is a record type, and $\{a = 12,\ b = 23.2\}$ is a record of $\{a : int,\ b : real\}$.

Record types are mainly used to specify classes. Thus, any record type for specifying classes has only abstract methods as its components. For example, the type for persons can be defined as follows:

$$
\begin{aligned}
\textbf{Type } Person = \{\ & name &&: \rightarrow string,\\
& age &&: \rightarrow int,\\
& younger &&: \textbf{thistype} \rightarrow bool,\\
& aging &&: int \rightarrow (\textbf{thistype})\\
\}\ &
\end{aligned}
$$

where **thistype** means type *Person*, and **(thistype)** means the internal type of type *Person*. As we will see, whenever type *Person* is inherited by other types, **thistype** will be changed to denote them.

Note that a basic type can be specified using a record type. For example, type *nat* can be defined using the following record type:

$$
\begin{aligned}
\textbf{Type } nat = \{\ & zero &&: \rightarrow \textbf{thistype},\\
& succ &&: \rightarrow \textbf{thistype},\\
& + &&: \textbf{thistype} \rightarrow \textbf{thistype},\\
& \times &&: \textbf{thistype} \rightarrow \textbf{thistype},
\end{aligned}
$$

....

}.

Similarly, pair types and other structured types can be specified using record types. Hence, any type can be considered as a record type.

3.1.3.2 Type Checking

Each object is declared to be of some class, and each class is regarded as an implementation of a given type. To type objects properly, we first need to type classes. As mentioned before, each class has a method scheme which is composed of method types. The type-checking of classes can be done through their method schemes.

Assume that there is a set of classes in which each class is a specific implementation of a type. Thus, each class in the set is naturally associated with a type. The assumption can be expressed as a function E that maps each class in the set to a corresponding type. E is called a *type assignment*. In addition, we extend E so that each class variable is associated with a type. We use $E' = E \cup \{\xi : \tau\}$ to denote the type assignment that is the same as E except that $E'(\xi) = \tau$ if ξ is not associated with any type in E. The following rules are used to type-check classes.

$$E \cup \{\xi : \tau\} \models \xi : \tau \tag{3.11}$$

$$E \cup \{c : \tau\} \models c : \tau \tag{3.12}$$

$$E \models \textbf{thisclass} : \textbf{thistype} \tag{3.13}$$

$$E \models (\textbf{thisclass}) : (\textbf{thistype}) \tag{3.14}$$

$$\frac{E \models \eta_1 : \tau_1,\ E \models \eta_2 : \tau_2}{E \models \eta_1 \hookrightarrow \eta_2 : \tau_1 \rightarrow \tau_2} \tag{3.15}$$

$$\frac{E \models \eta_1 : \tau_1, \cdots, E \models \eta_n : \tau_n}{E \models < \{\cdots\}, \{a_1 = e_1 : \eta_1, \cdots, a_n = e_n : \eta_n\} > : \{a_1 : \tau_1, \cdots, a_n : \tau_n\}} \tag{3.16}$$

The data portion of a class is not involved in type-checking classes. Hence, any two classes with the same method scheme have the same type.

Now objects can be associated with types. Intuitively, an object is of some type if there exists a class such that the object is an instance of the class and the class is an implementation of the type. The intuition can be formalized in the following rule:

$$\frac{A \models \vartheta : \xi,\ E \models \xi : \tau}{A, E \models \vartheta : \tau}. \tag{3.17}$$

The rule can be expanded by replacing ϑ, ξ, and τ with corresponding expressions. In general, we can type an expression using the following rules.

$$\frac{E \models \xi : \tau}{A, E \models \vartheta^{\xi} : \tau} \tag{3.18}$$

$$\frac{E \models \xi : \tau,\ A \models x : \xi}{A, E \models x : \tau} \tag{3.19}$$

$$\frac{A, E \models e_1 : \tau_1,\ \cdots,\ A, E \models e_n : \tau_n}{A, E \models <\{\cdots\}, \{a_1 = e_1,\ \cdots,\ a_n = e_n\}> : \{a_1 : \tau_1,\ \cdots,\ a_n : \tau_n\}} \tag{3.20}$$

$$\frac{A, E \models e : \{a_1 : \tau_1,\ a_2 : \tau_2,\ \cdots,\ a_n : \tau_n\}}{A, E \models e.a_i : \tau_i \quad (1 \leq i \leq n)} \tag{3.21}$$

$$\frac{A \cup \{x : \eta_1\} \models e : \eta_2,\ E \models \eta_1 : \tau_1,\ E \models \eta_2 : \tau_2}{A, E \models (\lambda x : \eta_1 . e) : \tau_1 \rightarrow \tau_2} \tag{3.22}$$

$$\frac{A, E \models e : \tau_1 \rightarrow \tau_2,\ A, E \models e' : \tau_1}{A, E \models e(e') : \tau_2} \tag{3.23}$$

3.1.4 Kinds

Different types may have common properties. A typical example is that many types have a total order, such as *nat*, *int*, *real*, and *string*. We say that all types that share a set of common properties form a *kind*. Obviously, kinds are entities which are one-level higher than types.

Consider words (strings) and acyclic graphs, where words can be implemented by arrays or linked lists and acyclic graphs by adjacency matrices or adjacency lists, etc. Obviously, words and acyclic graphs are two different types. Here, we name them type W and type G, respectively. These two different types have a common property: the distance between two words or two acyclic graphs can be measured [61]; i.e., they all have a distance function. Naturally, we represent the set of types that have a distance function by a kind called K. Clearly, both W and G are types in K. Let A be an algorithm that, for a given set of objects which have a distance function, produces the shortest distance between any two distinct objects of the set. Then, the domain of the algorithm can be defined with kind K. As we will argue, kinds are appropriate for defining algorithm domains.

3.1.4.1 Kind Definitions

The syntax of kinds is given below:

$$\kappa ::= Single(\tau) \mid Power(\tau) \mid TOP \mid \textbf{thiskind} \mid [id_1 : \kappa_1, id_2 : \kappa_2, \cdots, id_n : \kappa_n]$$

where **thiskind** is a special variable that denotes the kind being defined.

Singleton Kinds. Types are different from each other. Hence, any type τ constitutes a *singleton* kind, denoted $Single(\tau)$, which has only τ as its element.

Power Kinds. For any type τ, $Power(\tau)$ is a special kind that denotes the set of all subtypes of type τ. The concept of subtypes will be defined later. By using a type variable restricted to a power kind in a polymorphic function, bounded quantification becomes a special case of kind-bounded quantification as we will see.

Universal Kind. Any type satisfies the empty set of properties. Hence, all types form the *universal* kind that is defined by the empty set. We use TOP to denote the universal kind. As we will see, universal quantification is a special case of kind-bounded quantification since an unrestricted type variable is essentially a type variable restricted to TOP.

Function Kinds. A *function* kind is $\kappa_1 \mapsto \kappa_2$. It specifies a set of function types, each of which has an input type of κ_1 and an output type of κ_2. In other words, if τ_1 is a type of κ_1 and τ_2 is a type of κ_2, then $\tau_1 \rightarrow \tau_2$ is a function type of $\kappa_1 \mapsto \kappa_2$. For example, all unary function types satisfy the function kind $TOP \mapsto TOP$, and all binary function types satisfy the function kind $TOP \times TOP \mapsto TOP$.

Record Kinds. A *record* kind is represented as $[a_1 : \kappa_1, a_2 : \kappa_2, \cdots, a_n : \kappa_n]$, i.e., an unordered set of labelled kinds enclosed with "[" and "]". A record kind is mainly used to define the properties that are shared by a set of record types. For example, the following is a record kind for various types of stacks.

$$
\begin{aligned}
\textbf{Kind } STACK = [\, & push && : TOP^{\sim} \mapsto (\textbf{thiskind}^{\sim}), \\
& pop && : \ \mapsto (\textbf{thiskind}^{\sim}), \\
& top && : \ \mapsto TOP^{\sim}, \\
& empty && : \ \mapsto Single(bool) \\
]
\end{aligned}
$$

Here we use the symbol "$\sim$", which means that all occurrences of a kind that are marked with the superscript "$\sim$" have to be matched by the same type. The use of "$\sim$" is necessary since we would not allow a type of $STACK$ to have a ***push*** method accepting objects of type T_1 and a *top* method returning objects of T_2, where T_1 is different from T_2. A type T is of $STACK$ if it possesses methods ***push*** $: T' \rightarrow (T)$, $pop : \rightarrow (T)$, $top : \rightarrow T'$, and $empty : \rightarrow bool$ such that T' is a type of TOP. Similarly, all occurrences of a kind marked with the superscript "$\sim\sim$" have to be matched by the same type, and so on.

Record kinds can be nested; i.e., a record kind can be defined in terms of other record kinds. The nested structure of record kinds is appropriate for defining the common properties that are shared by complicated types. Accordingly, there is a selection operation on record kinds. Let $\kappa = [a_1 : \kappa_1,\ a_2 : \kappa_2, \cdots, a_n : \kappa_n]$. Then $\kappa.a_i$

is κ_i, $1 \leq i \leq n$.

3.1.4.2 Kind Checking

Now we provide the rules which are needed to check if a type is of a given kind. Kind-checking is based on *a kind assignment* G which restricts each type variable to a kind. Let $G' = G \cup \{t : \kappa\}$. Then G' is the same as G except that $G'(t) = \kappa$ (i.e., t is restricted to κ under G'). A type τ is said to be of a kind κ under G, denoted $G \models \tau : \kappa$, if $\tau : \kappa$ can be derived from G using the following rules.

Basic Rules

The following are basic rules which are used to kind-check types.

$$G \cup \{t : \kappa\} \models t : \kappa \tag{3.24}$$

$$G \models \tau : Single(\tau) \tag{3.25}$$

$$G \models \tau : Power(\tau) \tag{3.26}$$

$$G \models \tau : TOP \tag{3.27}$$

$$G \models \textbf{thistype} : \textbf{thiskind} \tag{3.28}$$

$$G \models (\textbf{thistype}) : (\textbf{thiskind}) \tag{3.29}$$

$$\frac{G \models \tau_1 : \kappa_1,\ G \models \tau_2 : \kappa_2}{G \models \tau_1 \rightarrow \tau_2 : \kappa_1 \mapsto \kappa_2} \tag{3.30}$$

$$\frac{G \models \tau_1 : \kappa_1,\ G \models \tau_2 : \kappa_2,\ \cdots,\ G \models \tau_n : \kappa_n}{G \models \{m_1 : \tau_1, m_2 : \tau_2, \cdots, m_n : \tau_n, \cdots\} : [m_1 : \kappa_1, m_2 : \kappa_2, \cdots, m_n : \kappa_n]} \tag{3.31}$$

Reordering

A record kind is an unordered set of labelled kinds. Hence, the order of components in a record kind is irrelevant. In other words, we can write down components of a record kind in any order. Thus, we have another rule to kind-check a record type:

$$\frac{G \models \tau_1 : \kappa_1, G \models \tau_2 : \kappa_2, \cdots, G \models \tau_n : \kappa_n}{G \models \{m_{i_1} : \tau_{i_1}, m_{i_2} : \tau_{i_2}, \cdots, m_{i_n} : \tau_{i_n}, \cdots\} : [m_1 : \kappa_1, m_2 : \kappa_2, \cdots, m_n : \kappa_n]} \tag{3.32}$$

where $(i_1,\ i_2,\ \cdots,\ i_n)$ is a permutation of $(1,\ 2,\ \cdots,\ n)$. For example, the following record kind is the same as $STACK$:

$$\begin{aligned}
\textbf{Kind } STACK_1 = [\ & push &&: TOP^{\sim} \mapsto (\textbf{thiskind}^{\sim}),\\
& empty &&: \ \mapsto Single(bool),\\
& top &&: \ \mapsto TOP^{\sim},\\
& pop &&: \ \mapsto (\textbf{thiskind}^{\sim})\\
].
\end{aligned}$$

Renaming

Different types may satisfy the same properties although they use different symbols for their operators. For example, we may use "$\leq$" to denote the comparison operator for integers, and use "*strcmp*" to denote the comparison operator for strings. Although "$\leq$" and "*strcmp*" are used for different types, both of them define a total order on the corresponding types.

A renaming operation is expressed as $[x \leftarrow y]$, where x and y denote two identifiers. Let τ be a record type which has a field labelled with x. If y does not occur as a label in τ, then $\tau[x \leftarrow y]$ is a record type that is the same as τ except that x is changed to y. For example, if $\tau = \{x_1 : \sigma_1,\ x_2 : \sigma_2\}$, then $\tau[x_1 \leftarrow y_1] = \{y_1 : \sigma_1,\ x_2 : \sigma_2\}$, and $\tau[x_1 \leftarrow y_1][x_2 \leftarrow y_2] = \{y_1 : \sigma_1,\ y_2 : \sigma_2\}$. Let γ denote a sequence of renaming operations. The following is another rule for kind-checking a record type:

$$\frac{G \models \tau_1 : \kappa_1,\ G \models \tau_2 : \kappa_2, \cdots, G \models \tau_n : \kappa_n}{G \models \{m_1 : \tau_1, m_2 : \tau_2, \cdots, m_n : \tau_n, \cdots\}\gamma : [m_1 : \kappa_1, m_2 : \kappa_2, \cdots, m_n : \kappa_n]}. \tag{3.33}$$

Similarly, we can apply a sequence of renaming operations to a record kind. Thus, we have that $[m_1 : \kappa_1, m_2 : \kappa_2, \cdots, m_n : \kappa_n]$ is the same as $[m_1 : \kappa_1, m_2 : \kappa_2, \cdots, m_n : \kappa_n]\gamma$.

3.2 Subclass, Subtype, and Subkind

The concept of ***subtypes*** has been associated with ***inheritance*** in many OOPLs, such as C++ [53], ***Eiffel*** [43], and ***Trellis/Owl*** [50]. That is, a type T_1 is a subtype of a type T_2 if T_1 is defined by inheriting from T_2. However, it has become clear recently that code inheritance does not automatically lead to subtype [20].

In this section, we develop a model of inheritance in the four-level hierarchy of entities. We introduce the concept of *subkind* to capture the inclusion relation between two kinds. Intuitively, if a kind κ_1 is a subkind of a kind κ_2, then any type of κ_1 is also a type of κ_2. Hence, in general, a subkind is a more restrictive specification than a superkind.

3.2.1 Subclass

Classes may be inherited to define other classes. In this way, we do not necessarily define new classes from scratch each time. Class inheritance can be expressed using the following syntax:

$$\textbf{Class}\ \xi = \{\xi_i\}_{1\le i\le p}\ \textbf{with} < \{a_1 : \eta_1, \cdots, a_m : \eta_m\}, \{b_1 = e'_1 : \eta'_1, \cdots, b_n = e'_n : \eta'_n\} >$$

which means that class ξ is defined by inheriting ξ_i, $1 \le i \le p$, and adding new data fields $\{a_i : \eta_i\}_{1\le i\le m}$ and method fields $\{b_i = e'_i : \eta'_i\}_{1\le i\le n}$. If a method b_i $(1 \le i \le n)$ has been defined in some class ξ_j $(1 \le j \le p)$, then the new definition of b_i will overlay the old definition of b_i. $\{\xi_i\}_{1\le i\le p}$ defines p ancestor classes of ξ. If $p > 1$, then multiple inheritance is realized.

Now we reconsider the class example for persons. An employee is also a person. Hence, the class for employees contains all fields of the class for persons. We do not need to define a class for employees from scratch. Instead, we can define one by inheriting from class *person* as follows:

$$\textbf{Class}\ employee = \{person\}\ \textbf{with} <$$
$$\{Salary : real\},$$

$$\begin{aligned}
\{\mathit{salary} &= \{\mathit{self.Salary}\} \\
&: \hookrightarrow \mathit{real}, \\
\mathit{add\_salary} &= \lambda s : \mathit{real}.\ \{\mathit{update}(\mathit{self}, \mathit{Salary}, \mathit{self.Salary} + s)\} \\
&: \mathit{real} \rightarrow (\textbf{thisclass}) \\
\}>
\end{aligned}$$

where all method and data fields of *person* are inherited unchanged by *employee*. But **thisclass** in *employee* denotes *employee* rather than *person*. Thus, the method *younger* in *employee* will accept objects of *employee* rather than objects of *person*.

According to the syntax of class inheritance, a subclass can have more than one superclass. For example, we first define another subclass of *person*, named *student*. Then the class *gradStudent* can be defined by inheriting from both *student* and *employee*.

$$\begin{aligned}
\textbf{Class}\ \mathit{student} = \{\mathit{person}\}\ \textbf{with} <& \\
&\{\mathit{Class} : \mathit{string}\}, \\
&\{\mathit{class} = \{\mathit{self.Class}\} \\
&\quad : \hookrightarrow \mathit{string}, \\
&\ \ \mathit{upgrade} = \lambda c : \mathit{string}.\{\mathit{update}(\mathit{self}, \mathit{Class}, c)\} \\
&\quad : \mathit{string} \rightarrow (\textbf{thisclass}) \\
&\}>
\end{aligned}$$

$$\begin{aligned}
\textbf{Class}\ \mathit{gradStudent} = \{\mathit{student}, \mathit{employee}\}\ \textbf{with} <& \\
&\{\mathit{Supervisor} : \mathit{string}\}, \\
&\{\mathit{supervisor} = \{\mathit{self.Supervisor}\} \\
&\quad : \hookrightarrow \mathit{string} \\
&\}>
\end{aligned}$$

Let $\mathcal{C}$ be a set of classes. A class ξ_1 is said to be a ***subclass*** of a class ξ_2 in $\mathcal{C}$, denoted $\mathcal{C} \models \xi_1 \leq^{(c)} \xi_2$, if $\xi_1 \leq^{(c)} \xi_2$ can be derived from $\mathcal{C}$ using the following rules:

- $\mathcal{C} \models \xi \leq^{(c)} \xi$ if class ξ is in $\mathcal{C}$;

- $\mathcal{C} \models \xi_1 \leq^{(c)} \xi_2$ if $\mathcal{C}$ has a class definition of the form
  Class $\xi_1 = \{\cdots, \xi_2, \cdots\}$ **with** $< \{\cdots\}, \{\cdots\} >$;
- $\mathcal{C} \models \xi_1(\bar{C}) \leq^{(c)} \xi_2(\bar{C})$ if $\mathcal{C} \models \xi_1(\bar{c}) \leq^{(c)} \xi_2(\bar{c})$, where $\xi_i(\bar{c})$ $(i = 1, 2)$ is a parameterized class, $\bar{c}$ is a list of class variables, and $\bar{C}$ is an instance of $\bar{c}$;
- $\mathcal{C} \models \xi_1 \leq^{(c)} \xi_3$ if $\mathcal{C} \models \xi_1 \leq^{(c)} \xi_2$ and $\mathcal{C} \models \xi_2 \leq^{(c)} \xi_3$.

Obviously, the subclass relation defines a partial order on the set of all classes. Since a subclass inherits all methods and data structures of a superclass, objects of a subclass can be used where those of a superclass are expected. However, a coercion is often needed from objects of a subclass to those of a superclass. For example, an object of *employee* can be treated as one of *person* if we ignore its data field *Salary* and method *add_salary*. We can express the relation between a subclass and a superclass with the following rule:

$$\frac{\mathcal{C} \models \xi_1 \leq^{(c)} \xi_2, \ \mathcal{A} \models \vartheta : \xi_1}{\mathcal{A}, \mathcal{C} \models \mathbf{coerce}_{\xi_1, \xi_2}(\vartheta) : \xi_2} \qquad (3.34)$$

where $\mathbf{coerce}_{\xi_1, \xi_2}$ denotes the coercion function from objects of ξ_1 to objects of ξ_2. The semantics of coercion will be defined in the framework of an object-oriented algebraic theory which will be introduced later.

By using class inheritance, we can define a number of subclasses from a given class, and from a subclass we can define a number of *sub*-subclasses, and so on. Hence, class inheritance supports code sharing among many classes. For example, we can define a class *secretary* from *employee*. Thus, the code defined in *person* is also shared by *employee* and *secretary*. Class inheritance also supports incremental software design. In each step of class inheritance, a superclass is specialized by redefining existing methods and adding new methods. Thus, classes become more and more specific after a number of steps of class inheritance. This process ends at a required class.

3.2.2 Subtype

Subtype was originally developed to deal with the inclusion relation between two types [14]. Intuitively, if type σ is a subtype of type τ, then any object of σ would be used where objects of τ are expected. In this subsection, we give the definition of subtype in our four-level type hierarchy.

3.2.2.1 Subtype Checking

Let R be a set of subtype relations. $R \cup \{\sigma \leq^{(t)} \tau\}$ is the same as R except that σ is restricted to be a subtype of τ. Formally, type τ_1 is said to be a ***subtype*** of type τ_2 under R, denoted $R \models \tau_1 \leq^{(t)} \tau_2$, if the relation $\tau_1 \leq^{(t)} \tau_2$ can be derived from R using the following rules.

Trivial Rules. Any type is a subtype of itself. A type variable is also regarded as a subtype of itself. Thus, we have the following trivial rules:

$$R \models \tau \leq^{(t)} \tau, \qquad R \models t \leq^{(t)} t. \tag{3.35}$$

In addition, if τ_1 is a subtype of τ_2 and τ_2 is a subtype of τ_3, then τ_1 is also a subtype of τ_3. Thus, we have the following rule:

$$\frac{R \models \tau_1 \leq^{(t)} \tau_2, \ R \models \tau_2 \leq^{(t)} \tau_3}{R \models \tau_1 \leq^{(t)} \tau_3}. \tag{3.36}$$

Obviously, $\leq^{(t)}$ defines a partial order on the set of types.

Rule for Function Types. The subtype relation also extends to function types. A function type $\sigma_1 \to \tau_1$ is a subtype of a function type $\sigma_2 \to \tau_2$ if σ_2 is a subtype of σ_1 and τ_1 is a subtype of τ_2. Thus, any function working on type σ_1 can be regarded as a function working on a subtype σ_2 of σ_1, and any function returning values of type τ_1 can be regarded as a function returning values of a supertype τ_2 of τ_1. We have the following rule to determine the subtype relation between two function types:

$$\frac{R \models \sigma_2 \leq^{(t)} \sigma_1, \ R \models \tau_1 \leq^{(t)} \tau_2}{R \models \sigma_1 \to \tau_1 \leq^{(t)} \sigma_2 \to \tau_2} \tag{3.37}$$

where the direction of $\sigma_2 \leq^{(t)} \sigma_1$ is opposite to that of $\tau_1 \leq^{(t)} \tau_2$, which is called *contravariant*.

Rules for Record Types. Let σ and τ be two record types. If all fields of τ are also present in σ, then σ is a subtype of τ, which can be expressed as follows:

$$R \models \{a_1 : \sigma_1,\ \cdots,\ a_n : \sigma_n,\ \cdots,\ a_{n+m} : \sigma_{n+m}\} \leq^{(t)} \{a_1 : \sigma_1,\ \cdots,\ a_n : \sigma_n\}. \quad (3.38)$$

This rule can be generalized so that τ is a subtype of σ if τ has all the fields of σ (possibly more) and the common fields of τ and σ are in the subtype relation. This generalization can be expressed in the following rule:

$$\frac{R \models \sigma_1 \leq^{(t)} \tau_1,\ \cdots,\ R \models \sigma_n \leq^{(t)} \tau_n}{R \models \{a_1 : \sigma_1, \cdots, a_n : \sigma_n,\ \cdots, a_{n+m} : \sigma_{n+m}\} \leq^{(t)} \{a_1 : \tau_1, \cdots, a_n : \tau_n\}}. \quad (3.39)$$

3.2.2.2 Inheritance and Subtype

Each class has a type. However, the subclass relation between two classes does not mean that there must be a subtype relation between their types. Consider the type *Person* for class *person* that is defined in the previous section and the type *Employee* for class *employee* that is defined as follows:

$$\begin{aligned}
\textbf{Type}\ Employee = \{\ & name && : \ \rightarrow string,\\
& age && : \ \rightarrow int,\\
& salary && : \ \rightarrow real,\\
& younger && : \textbf{thistype} \rightarrow bool,\\
& aging && : int \rightarrow (\textbf{thistype}),\\
& add\_salary && : real \rightarrow (\textbf{thistype})\\
\}.&
\end{aligned}$$

According to the subtyping rules, *Employee* is a subtype of *Person* if the function type of *younger* (:**thistype** $\rightarrow$ *bool*) in *Employee* is a subtype of the function type of *younger* (:**thistype** $\rightarrow$ *bool*) in *Person*. We replace the two occurrences of **thistype** with *Employee* and *Person*, respectively. It is obvious that *Employee* is a subtype of *Person* if *Person* is a subtype of *Employee* because of the contravariance. This is a contradiction. Therefore, *Employee* is not a subtype of *Person*.

We can show that inheritance is not subtype in another way. Instead of defining types from scratch each time, we can define new types by inheriting from other types. The following is the syntax of type inheritance:

$$\textbf{Type } \tau = \{\tau_1, \cdots, \tau_n\} \textbf{ with } \{a_1 : \tau_1', \cdots, a_m : \tau_m'\}$$

which means that τ inherits from $\tau_1, \cdots, \tau_n$ plus $a_i : \tau_i'$, $1 \leq i \leq m$. Type τ is said to be an *heir* type of τ_i, and τ_i is said to be an *ancestor* type of τ, $1 \leq i \leq n$. *Employee* is an heir type of *Person* since it can also be defined as follows:

$$\begin{aligned}\textbf{Type } \mathit{Employee} = \{\mathit{Person}\} \textbf{ with } \{ & \\ & \mathit{salary} \quad : \ \rightarrow \mathit{real}, \\ & \mathit{add\_salary} \ : \ \mathit{real} \rightarrow (\textbf{thistype}) \\ & \}\end{aligned}$$

where all occurrences of **thistype** mean *Employee*. However, *Employee* is not a subtype of *Person*. Hence, an heir type is not necessarily a subtype of an ancestor type.

3.2.3 Subkind

Subkind is used to deal with the inclusion relation between two kinds. Intuitively, if a kind κ is a subkind of a kind κ', then any type of κ is also a type of κ'. Hence, a subkind would be a more restrictive specification of types than a superkind.

3.2.3.1 Subkind Checking

Let W be a set of subkind restrictions. Kind κ_1 is said to be a ***subkind*** of kind κ_2 under W, denoted $W \models \kappa_1 \leq^{(k)} \kappa_2$, if $\kappa_1 \leq^{(k)} \kappa_2$ can be derived from W using the following rules.

Trivial Rules. For any type τ, $Single(\tau)$ is a subkind of $Power(\tau)$ and $Power(\tau)$ is a subkind of Top. Hence, we have the following trivial rules:

$$W \models Single(\tau) \leq^{(k)} Power(\tau) \tag{3.40}$$

$$W \models Power(\tau) \leq^{(k)} TOP. \tag{3.41}$$

Let $TYPE^{\kappa}$ denote the set of types that are of kind κ. It is obvious that $TYPE^{Single(\tau)} \subseteq TYPE^{Power(\tau)}$ and $TYPE^{Power(\tau)} \subseteq TYPE^{TOP}$ for any type τ.

Let κ_1, κ_2, and κ_3 be three kinds. If κ_1 is a subkind of κ_2 and κ_2 is a subkind of κ_3, then κ_1 is a subkind of κ_3. We have the following rule:

$$\frac{W \models \kappa_1 \leq^{(k)} \kappa_2,\ W \models \kappa_2 \leq^{(k)} \kappa_3}{W \models \kappa_1 \leq^{(k)} \kappa_3}. \tag{3.42}$$

In addition, any kind κ is a subkind of itself, which can be expressed as the following rule:

$$W \models \kappa \leq^{(k)} \kappa. \tag{3.43}$$

Thus, "$\leq^{(k)}$" is a reflexive and transitive relation over kinds.

Rule for Function Kinds. The subkind relation extends to function kinds. A function kind $\kappa_1 \mapsto \kappa_2$ is a subkind of a function kind $\kappa_1' \mapsto \kappa_2'$ if κ_1 is a subkind of κ_1' and κ_2 is a subkind of κ_2'. This definition can be expressed as the following rule:

$$\frac{W \models \kappa_1 \leq^{(k)} \kappa_1',\ W \models \kappa_2 \leq^{(k)} \kappa_2'}{W \models \kappa_1 \mapsto \kappa_2 \leq^{(k)} \kappa_1' \mapsto \kappa_2'}. \tag{3.44}$$

For example, both $TOP \mapsto Single(real)$ and $Single(real) \mapsto TOP$ are subkinds of $TOP \mapsto TOP$. Obviously, a function type of $\kappa_1 \mapsto \kappa_2$ is also a function type of $\kappa_1' \mapsto \kappa_2'$.

Rules for Record Kinds. A record kind is a property specification for a set of types. It can be made more restrictive by adding more fields, and, conversely, it can be made less restrictive by eliminating some fields. Hence, a type satisfying a more restrictive record kind is also a type satisfying a less restrictive record kind. For example, the following is a record kind which is less restrictive than $STACK$.

$$\begin{aligned}\textbf{Kind } STACK_2 = [\ & pop &&: \mapsto (\textbf{thiskind}),\\ & top &&: \mapsto TOP,\\ & empty &&: \mapsto Single(bool)\\ ]&&&\end{aligned}$$

Obviously, any type satisfying $STACK$ is also a type satisfying $STACK_2$.

A record kind κ is a subkind of a record kind κ' if κ has all the fields of κ' (possibly more). The following rule is used to check the subkind relation between two record kinds:

$$W \models [m_1 : \kappa_1, \cdots, m_n : \kappa_n, \cdots, m_{n+k} : \kappa_{n+k}] \leq^{(k)} [m_1 \cdot \kappa_1, \cdots, m_n : \kappa_n]. \qquad (3.45)$$

The rule can be generalized so that a record kind κ_1 is a subkind of a record kind κ_2 if their corresponding fields are in the subkind relation. That is,

$$\frac{W \models \kappa_1 \leq^{(k)} \kappa'_1, \cdots, W \models \kappa_n \leq^{(k)} \kappa'_n}{W \models [m_1 : \kappa_1, \cdots, m_n : \kappa_n, \cdots, m_{n+k} : \kappa'_{n+k}] \leq^{(k)} [m_1 : \kappa'_1, \cdots, m_n : \kappa'_n]}. \qquad (3.46)$$

3.2.3.2 Kind Inheritance

Similar to type inheritance, we can define new record kinds by inheriting from other record kinds. The following is the syntax of kind inheritance:

$$\textbf{Kind } \kappa = \{\kappa_1, \cdots, \kappa_n\} \textbf{ with } [a_1 : \kappa'_1, \cdots, a_m : \kappa'_m]$$

where κ is defined from κ_i, $1 \leq i \leq n$, by adding $a_j : \kappa'_j$, $1 \leq j \leq m$. That is, κ includes all the fields defined in κ_i, $1 \leq i \leq n$, plus a_j, $1 \leq i \leq m$. Hence, an heir kind is also a subkind of an ancestor kind.

3.3 Remark

We have established a four-level hierarchy of entities: objects, classes, types, and kinds. Objects are related to types indirectly. That is, an object ϑ is of a type τ if there exists a class ξ such that ϑ is an object of ξ and ξ is an implementation of τ. Thus, if an object variable is declared to be of a given type, it can carry objects of any class that implements the type. Similarly, classes are associated with kinds indirectly. A class ξ is said to be of a kind κ if there exists a type τ such that ξ is a class of τ and τ is a type in κ.

In this hierarchy, each entity at a given level (except the object level) is a set of entities at the next lower level. Hence, entities at a higher level of the hierarchy are also at a higher abstraction level than entities at a lower level of the hierarchy. In particular, kinds are at the highest abstraction level. Classes are implementation-dependent entities. Hence, they are not suited for defining the domains of algorithms. Types can be used to define the domains of algorithms to some extent since a type defines a number of classes. However, types are too restrictive since all classes of a type have to have exactly the same external behaviour. A kind is defined with a set of implementation-independent properties which are satisfied by a group of types. These properties are also independent of any specific type. Hence, kinds would be used to define the domains of algorithms.

In fact, an algorithm demands only certain properties on each type of its domain and it does not require these properties to be of a certain specific type. Also, two or more parameters of an algorithm may be of different types that satisfy the same set of properties. All of these characteristics can be properly defined only with kinds. Hence, only kinds can be used to define the domains of algorithms properly. Later we will provide examples to demonstrate how kinds are used in defining language-supported algorithms.

Kinds cannot be replaced by supertypes. Consider type W for words (strings) and type G for acyclic graphs again. Both W and G are types in kind K which is the set of all types that have a distance function. If we define a common supertype

T for both W and G using the distance function as a property, this would imply that there is a distance function between a word and an acyclic graph by the subtyping principle. This is clearly not what we intend to define. In other words, there does not exist a common supertype T for W and G if we assume that the distance between a word and an acyclic graph is meaningless.

Kinds are also more appropriate for specifying the properties shared by types than parameterized types. First of all, we would use a higher-level entity to represent a set of types sharing common properties. However, parameterized types are treated as a special kind of types rather than entities at a higher level. Secondly, there exist such kinds that cannot be expressed by parameterized types. For example, we do not have a parameterized type to specify all types of stacks that have person objects as elements. In our approach, these types would be specified by the following two kinds:

$$
\begin{array}{llll}
\textbf{Kind } Human = [& name & : & \mapsto Single(string), \\
 & younger & : & \textbf{thiskind} \mapsto Single(bool) \\
] & & &
\end{array}
$$

$$
\begin{array}{llll}
\textbf{Kind } PSTACK = [& push & : & HUMAN^{\sim} \mapsto (\textbf{thiskind}^{\sim}), \\
 & pop & : & \mapsto (\textbf{thiskind}^{\sim}), \\
 & top & : & \mapsto HUMAN^{\sim}, \\
 & empty & : & \mapsto Single(bool) \\
]. & & &
\end{array}
$$

Chapter 4

Polymorphism and Algorithmic Abstraction

In this chapter, we detail various forms of polymorphism that can be established in the previously defined type hierarchy. These include ***universal-bounded*** polymorphism, ***membership-bounded*** polymorphism, and ***inclusion-bounded*** polymorphism.

Different forms of polymorphism are specified with different forms of quantification on type (class) variables. Universal quantification can be used to introduce unrestricted type (class) variables. Besides universal quantification, there are the following forms of quantification:

$\forall c \in^{(c)} \tau$	type-bounded quantification, which restricts class variable c to type τ;
$\forall t \in^{(t)} \kappa$ --	kind-bounded quantification, which restricts type variable t to kind κ;
$\forall c \preceq^{(c)} \xi$ --	subclass-bounded quantification, which restricts class variable c to the set of all subclasses of class ξ;
$\forall t \leq^{(t)} \tau$ -	subtype-bounded quantification, which restricts type variable t to the set of all subtypes of type τ.

Note that a type can be regarded as a set of classes. Hence, it makes sense that class variable c being restricted to type τ is denoted by $\forall c \in^{(c)} \tau$. Similarly, $\forall t \in^{(t)} \kappa$ denotes that type variable t ranges over the set of all types specified by κ.

At the end of this chapter, we clarify the relation between polymorphism and algorithmic abstraction.

4.1 Universal-Bounded Polymorphism

Universal-bounded polymorphism is the simplest form of polymorphism that has been used in programming languages. Examples of universal-bounded polymorphism include *templates* in C++ [53], *generic functions* in *Ada* [3], and ***polymorphic functions*** in *ML* [56]. Universal-bounded polymorphism supports the reuse of a function on all classes (types) without any change to its function body (at the source code level).

Universal-bounded polymorphism is realized using universal quantification. There are two forms of universal quantification, which introduce unrestricted class variables and unrestricted type variables, respectively. Hence, there are two forms of universal-bounded polymorphism in our four-level hierarchy of entities.

4.1.1 *Form 1* Universal-Bounded Polymorphism

In *Form 1* universal-bounded polymorphism, a function is defined so that it is applicable to all classes. Its use can be illustrated with the following function:

$$\begin{aligned} \textbf{Fun } id_1 &= (\forall c).\lambda x : c.\ \underline{x} \\ id_1 &: (\forall c).c \rightarrow c \end{aligned}$$

where c is an universal-bounded class variable that ranges over the set of all classes. Besides the class variable c, the function id_1 also includes an object variable x that is restricted to c. Hence, any actual call to id_1 needs two actual parameters: one is a class, and the other is an object of the class. Since c can be any class, id_1 defines a universal *identity* function. A function in *Form 1* universal-bounded polymorphism is typed according to the following rule:

$$\frac{A \models e : \eta}{A \models (\forall c).e : (\forall c).\eta} \tag{4.1}$$

where A is a class assignment defined in the previous chapter, c is a class variable which is not free in A, e is an expression, and η is a method type.

Since c is a class variable, id_1 can also be considered as a function from classes to functions. In other words, it generates an identity function from ξ to ξ for any class

ξ. For example, when id_1 is called with class *person*, an identity function over *person* is formed as follows:

$$\begin{aligned}\textbf{Fun } id_1(person) &= \lambda x : person.\ \underline{x} \\ id_1(person) &: person \rightarrow person.\end{aligned}$$

We use the following rule to type the one-step application of a function in *Form 1* universal-bounded polymorphism:

$$\frac{A \models e : (\forall c).\eta}{A \models e(\xi) : \{\xi/c\}\eta} \tag{4.2}$$

where $\{\xi/c\}\eta$ is a new method type that is obtained from η by substituting ξ for each occurrence of c in η.

4.1.2 *Form 2* Universal-Bounded Polymorphism

In *Form 2* universal-bounded polymorphism, we can define a function that is applicable to all types. This is realized by using a universal-bounded type variable in a function. For example, the identity function over all types can be defined as follows:

$$\begin{aligned}\textbf{Fun } id_2 &= (\forall t).(\forall c \in^{(c)} t).\lambda x : c.\ \underline{x} \\ id_2 &: (\forall t).(\forall c \in^{(c)} t).c \rightarrow c\end{aligned}$$

where t is a type variable restricted to all types by $(\forall t)$, and c is a class variable restricted to t by $(\forall c \in^{(c)} t)$. In fact, $(\forall c \in^{(c)} t)$ introduces another form of polymorphism, called *type-bounded* polymorphism, which will be detailed in the next section. The type of a function in *Form 2* universal-bounded polymorphism is based on the following rule:

$$\frac{A, E \models e : \eta}{A, E \models (\forall t).e : (\forall t).\eta} \tag{4.3}$$

where E is a type assignment defined in the previous chapter, and t is a type variable which is not free in A and E.

Note that any class is supposed to be an implementation of a type. Hence, id_2 is equivalent to id_1 since they have not only the same function body but also the same domain of classes. Thus, id_1 can be regarded as a simplified form of id_2. However, a function in *Form 2* universal-bounded polymorphism cannot be reduced to one in *Form 1* universal-bounded polymorphism if it uses two or more class variables that are restricted to be of the same type. In other words, *Form 1* universal-bounded polymorphism can only be used to define such functions whose object variables can be of any class.

A function in *Form 2* universal-bounded polymorphism can be regarded as a function from types to type-bounded functions. The one-step application of a function in *Form 2* universal-bounded polymorphism is typed according to the following rule:

$$\frac{A, E \models e : (\forall t).\eta}{A, E \models e(\sigma) : \{\sigma/t\}\eta}. \tag{4.4}$$

For example, when id_2 is called with type $Person$, an identity function over all classes of $Person$ is formed as follows:

$$\begin{aligned} \textbf{Fun } id_2(Person) &= (\forall c \in^{(c)} Person).\lambda x : c.\ \underline{x} \\ id_2(Person) &: (\forall c \in^{(c)} Person).c \to c. \end{aligned}$$

4.1.3 Remark

A fatal problem of universal-bounded polymorphism is that it cannot express any restriction on a class variable or a type variable which is used in a polymorphic function. Thus, improper classes or types cannot be excluded from the valid classes or types for a parameter of a polymorphic function. Consider the following function in *Form 1* universal-bounded polymorphism:

$$\begin{aligned} \textbf{Fun } double &= (\forall c).\lambda x : c.\ \underline{x + x} \\ double &: (\forall c).c \to c. \end{aligned}$$

It is obvious that *double* can apply only to such classes that have method "+". If we apply *double* to objects of some class that does not have method "+", an unexpected result will be caused. For example, we cannot apply ***double*** to objects of class ***person***.

In fact, there are very few properties that are shared by all classes or all types. Hence, universal-bounded polymorphism does not have a wide range of usage.

In order to prevent improper classes (types) from being supplied to a polymorphic function, we need some way to explicitly specify a range of classes (types) that satisfy certain conditions. Such a specification will not only provide a clear guide for users to use a polymorphic function but also allow the compiler to catch improper uses.

4.2 Membership-Bounded Polymorphism

Membership-bounded polymorphism is another form of polymorphism. In membership-bounded polymorphism, the domain of a function is explicitly restricted to a type or a kind. We can distinguish two forms of membership-bounded polymorphism: ***type-bounded*** polymorphism and *kind-bounded* polymorphism.

4.2.1 Type-Bounded Polymorphism

In *type-bounded* polymorphism, which is realized using type-bounded quantification, we can define a function that is explicitly restricted to all classes of a given type. The following is a type-bounded function that sorts an array of objects of a class of type *Person* using the bubble sort algorithm. The order of these objects is defined according to the following rules: (1) the object of a younger person precedes the object of an older person; (2) if two objects are of the same age, their orders are determined by the alphabetical orders of their names.

```
Fun $sort_1 = (\forall c \in^{(c)} Person).\lambda x : array\ of\ c.\{$
            $n = size(x);$
            for $i = 1$ to $n - 1$ do
                for $j = 1$ to $n - i$ do {
                    $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1],\ x[j])$
                    $else\ if\ x[j].younger(x[j+1])\ then$ continue
                    $else\ if\ (x[j+1].name() \leq x[j].name())$
                        $then\ swap(x[j+1],\ x[j])$
                };
            }
    $sort_1 : (\forall c \in^{(c)} Person).array\ of\ c \rightarrow array\ of\ c$
```

where "$\leq$" is the comparison operator over strings, and *size* and *swap* are two pre-defined routines. The routine *size* returns the size of an array, and the routine *swap* exchanges its two parameters. The class domain of $sort_1$ is explicitly declared to be the set of all classes of type *Person*. Hence, any call to $sort_1$ without a class of type *Person* will result in an unexpected result.

A type-bounded function is typed according to the following rule:

$$\frac{A, E \cup \{c : \tau\} \models e : \eta}{A, E \models (\forall c \in^{(c)} \tau).e : (\forall c \in^{(c)} \tau).\eta}. \tag{4.5}$$

Since a type-bounded function has a class variable restricted to a type, it can be used as a function from classes to functions. For example, the call to $sort_1$ with class *person* will result in the following function:

```
Fun $sort_1(person) = \lambda x : array\ of\ person.\{$
        $n = size(x)$;
        for $i = 1$ to $n - 1$ do
            for $j = 1$ to $n - i$ do {
                $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1],\ x[j])$
                $else\ if\ x[j].younger(x[j+1])\ then$ continue
                $else\ if\ (x[j+1].name() \leq x[j].name())$
                    $then\ swap(x[j+1],\ x[j])$
            };
        }
    $sort_1(person) : array\ of\ person \rightarrow array\ of\ person.$
```

The following rule is used to type the one-step application of a type-bounded function:

$$\frac{A, E \models \epsilon : (\forall c \in^{(c)} \tau).\eta, \;\; E \models \xi : \tau}{A, E \models \epsilon(\xi) : \{\xi/c\}\eta}. \tag{4.6}$$

Now we consider the function *double* introduced in the previous section. Obviously, *int* is a type that has "+". Hence we have a specific successor function for all classes of *int* as follows:

$$\begin{aligned} \textbf{Fun } double_1 &= (\forall c \in^{(c)} int).\lambda x : c.\ \underline{x + x} \\ double_1 &: (\forall c \in^{(c)} int).c \rightarrow c. \end{aligned}$$

Note that *nat* is also a type with "+". However, $double_1$ cannot be applied to classes of *nat* due to the restriction ($\forall c \in^{(c)} int$). Hence, we need a more precise way to specify the class domain of *double*.

4.2.2 Kind-Bounded Polymorphism

Types are different from each other. However, a group of types may share certain common properties. For example, all the objects of either *int* or *string* form a totally-ordered set. This fact suggests that a function defined for a specific type may be generalized to one that is applicable to a number of types sharing a common set of properties.

An example function is the one that selects the larger number between two integers. Obviously, "$\leq$"(comparison) is the only type-specific operator in this function. If we ignore the specific meaning of "$\leq$", the body of this function can be used for any type that has "$\leq$". These types that have "$\leq$" can be specified with the following kind:

$$\textbf{Kind } TotalOrder = [\ \leq : \textbf{thiskind} \mapsto Single(bool)\ ].$$

That is, a type T is of *TotalOrder* if it has $\leq$: $T \rightarrow bool$. Based on this kind, a polymorphic maximum function can be defined as follows:

$$\textbf{Fun } max = (\forall t \in^{(t)} TotalOrder).(\forall c \in^{(c)} t).\lambda x, y : c.\ \{$$

```
                                    if (x. ≤ (y)) then y else x
                                    }
    max  : (∀t ∈^(t) TotalOrder).(∀c ∈^(c) t).c × c → c
```

where type variable t is restricted to all types of *TotalOrder* by using the kind-bounded quantification $(\forall t \in^{(t)} TotalOrder)$. Such a function is called a ***kind-bounded*** function, which is typed according to the following rule:

$$\frac{A,\ E,\ G \cup \{t : \kappa\} \models e : \eta}{A,\ E,\ G \models (\forall t \in^{(t)} \kappa).e : (\forall t \in^{(t)} \kappa).\eta} \tag{4.7}$$

where G is a kind assignment which is defined in the previous chapter.

A kind-bounded function can be regarded as a function from types to type-bounded functions since a type variable is explicitly involved. In other words, any call to a kind-bounded function with a proper type will result in a type-bounded function. For example, *int* is a type of *TotalOrder*, and, therefore, the call to *max* with *int* will generate the following type-bounded function:

```
    Fun max(int) = (∀c ∈^(c) int).λx, y : c. {
                                    if (x. ≤ (y)) then y else x
                                    }
    max(int) : (∀c ∈^(c) int).c × c → c.
```

The following rule is used to type the one-step application of a kind-bounded function:

$$\frac{A, E, G \models e : (\forall t \in^{(t)} \kappa).\eta,\ G \models \sigma : \kappa}{A, E, G \models e(\sigma) : \{\sigma/t\}\eta}. \tag{4.8}$$

Note that we can also apply *max* to two objects ϑ_1 and ϑ_2 directly. In this case, we need a type τ of *TotalOrder* as well as a class ξ of τ such that ϑ_1 and ϑ_2 are two objects of ξ.

Consider the function *double* again. To reuse its function body to other proper types, we define a kind called *DOUBLE* as follows:

Kind $DOUBLE = [\ + : \textbf{thiskind}^{\sim} \mapsto \textbf{thiskind}^{\sim}\ ]$.

A type T is of *DOUBLE* if it has an abstract method "+" ($:T \rightarrow T$), probably different names. Then *double* can be rewritten as follows:

$$\begin{aligned}\textbf{Fun}\ double_2 &= (\forall t \in^{(t)} DOUBLE).(\forall c \in^{(c)} t).\lambda x : c.\ \underline{x + x}\\ double_2 &: (\forall t \in^{(t)} DOUBLE).(\forall c \in^{(c)} t).c \rightarrow c.\end{aligned}$$

Now $double_2$ can be applied to not only classes of *int* but also classes of any type, such as *nat* and *real*, that satisfies the properties specified in kind *DOUBLE*.

4.2.2.1 Simplification

In a kind-bounded function, a type variable is usually needed between a kind and a class variable since a class is not directly associated with a kind. However, in some situations, a kind-bounded function can be simplified by introducing the following syntactic abbreviation:

$$\xi \in^{(c)}: \kappa \longleftrightarrow (\exists \tau : TOP)(\xi \in^{(c)} \tau \wedge \tau \in^{(t)} \kappa).$$

That is, ξ is regarded as a class of kind κ if and only if there exists a type τ such that ξ is a class of τ and τ is a type of κ. For example, *max* can be simplified as follows:

$$\begin{aligned}\textbf{Fun}\ max' &= (\forall c \in^{(c)}: TotalOrder).\lambda x, y : c.\ \{\\ &\qquad if\ (x. \leq (y))\ then\ y\ else\ x\\ &\qquad \}\\ max' &: (\forall c \in^{(c)}: TotalOrder).c \times c \rightarrow c.\end{aligned}$$

Obviously, simplification makes a kind-bounded function more readable. Similarly, $double_2$ can be simplified as follows:

$$\begin{aligned}\textbf{Fun}\ double_2' &= (\forall c \in^{(c)}: DOUBLE).\lambda x : c.\ \underline{x + x}\\ double_2' &: (\forall c \in^{(c)}: DOUBLE).c \rightarrow c.\end{aligned}$$

However, a kind-bounded function cannot be simplified if it has two or more class variables that are required to be of the same type. In other words, the simplified form of kind-bounded quantification can only be used to define such functions in which a class variable can be instantiated by any class of a given kind.

A simplified kind-bounded function can be regarded as a function from classes to concrete functions. The following rule is used to type the one-step application of a simplified kind-bounded function:

$$\frac{A, E, G \models e : (\forall c \in^{(c)}: \kappa).\eta,\ G \models \sigma : \kappa,\ E \models \xi : \sigma}{A, E, G \models e(\xi) : \{\xi/c\}\eta}. \tag{4.9}$$

Let C denote a class whose type is *TotalOrder*. Then the call to *max'* with C will generate the following function:

Fun $max'(C) = \lambda x, y : C.\ \{\ if\ (x. \leq (y))\ then\ y\ else\ x\ \}$

$max'(C) : C \times C \rightarrow C.$

4.2.3 Remark

In kind-bounded polymorphism, a polymorphic function is represented as one that is applicable to all types of a kind. All types of a kind share a set of common properties. Hence, a kind-bounded function will have a consistent behaviour on all proper types. This fact indicates that kind-bounded polymorphism can be used to represent language-supported algorithms.

Universal quantification and subtype-bounded quantification can be realized using "$\forall t \in^{(t)} TOP$" and "$\forall t \in^{(t)} Power(T)$", respectively. Hence, universal-bounded polymorphism and subtype-bounded polymorphism (defined in the next section) are only two special cases of kind-bounded polymorphism. Type-bounded polymorphism is also subsumed by kind-bounded polymorphism since any type constitutes a singleton kind. Hence, kind-bounded polymorphism is the most general form of polymorphism.

4.3 Inclusion-Bounded Polymorphism

Inclusion-bounded polymorphism allows us to write a function that is applicable to all subtypes of a type or all subclasses of a class. Thus, there are two forms of inclusion-bounded polymorphism in our type hierarchy: ***subclass-bounded*** polymorphism and ***subtype-bounded*** polymorphism.

4.3.1 Subclass-Bounded Polymorphism

Subclass-bounded polymorphism is achieved when a function has a class variable that is explicitly restricted to the set of all subclasses of a given class. Such a function, called a ***subclass-bounded*** function, is applicable to all subclasses of the involved class. For example, the following is a subclass-bounded function that sorts an array of objects of class *person* (or any of its subclasses) based on the rule used in $sort_1$:

```
Fun $sort_2 = (\forall c \leq^{(c)} person).\lambda x : array\ of\ c.\{$
          $n = size(x)$;
          for $i = 1$ to $n - 1$ do
              for $j = 1$ to $n - i$ do {
                  $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1],\ x[j])$
                  $else\ if\ x[j].younger(x[j+1])\ then$ continue
                  $else\ if\ (x[j+1].name() \leq x[j].name())$
                      $then\ swap(x[j+1],\ x[j])$
              };
          }
     $sort_2 : (\forall c \leq^{(c)} person).array\ of\ c \rightarrow array\ of\ c$
```

where the condition $(\forall c \leq^{(c)} person)$ indicates that class variable c has to be instantiated by class *person* or any of its subclasses. Whenever c is instantiated by a subclass of *person*, x has to be an array of objects of the subclass. A subclass-bounded function is typed according to the following rule:

$$\frac{A,\ \mathcal{C} \cup \{c \leq^{(c)} \xi\} \models e : \eta}{A,\ \mathcal{C} \models (\forall c \leq^{(c)} \xi).e : (\forall c \leq^{(c)} \xi).\eta} \tag{4.10}$$

where $\mathcal{C}$ is a set of subclass restrictions (defined in the previous chapter).

As mentioned before, any object of a subclass can be coerced into one of a superclass. Hence, the above function could also be defined as follows:

```
Fun $sort'_2 = \lambda x : array\ of\ person.\{$
        $n = size(x)$;
        for $i = 1$ to $n - 1$ do
            for $j = 1$ to $n - i$ do {
                $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1],\ x[j])$
                $else\ if\ x[j].younger(x[j+1])\ then$ continue
                $else\ if\ (x[j+1].name() \leq x[j].name())$
                    $then\ swap(x[j+1],\ x[j])$
            };
        }
    $sort'_2 : array\ of\ person \rightarrow array\ of\ person.$
```

However, there is a subtle difference between *sort*$_2$ and *sort*$'_2$. That is, *sort*$'_2$ always accepts an array of objects of *person*. In other words, when *sort*$'_2$ is applied to an array of objects which belong to a subclass of *person*, these objects will be coerced into ones which contain only the fields defined in *person*. As a result, only truncated objects are included in the array returned by *sort*$'_2$. In contrast, *sort*$_2$ does not truncate the objects in an array parameter since the class of these objects is also provided to *sort*$_2$. The difference between *sort*$_2$ and *sort*$'_2$ can be expressed in the following rules:

$$\frac{A,\ \mathcal{C} \models e : (\forall c \leq^{(c)} \xi).\eta,\ \mathcal{C} \models \xi' \leq^{(c)} \xi}{A,\ \mathcal{C} \models e(\xi') : \{\xi'/c\}\eta} \tag{4.11}$$

$$\frac{A \models e : \xi_1 \hookrightarrow \xi_2,\ \mathcal{C} \models \xi' \leq^{(c)} \xi_1,\ A \models \vartheta : \xi'}{A,\ \mathcal{C} \models e(\mathbf{coerce}_{\xi',\xi_1}(\vartheta)) : \xi_2} \tag{4.12}$$

where $\mathbf{coerce}_{\xi',\xi_1}$ is the coercion function from class ξ' to class ξ_1, which is defined in the previous chapter.

Note that $sort_1$ is different from $sort_2$ although they have the same function body. That is, $sort_1$ is applicable to all classes of type *Person*, and $sort_2$ is applicable to all subclasses of class *person*.

4.3.2 Subtype-Bounded Polymorphism

Objects of a subtype may be used where objects of a supertype are expected. For example, we can rewrite $sort_1$ in subtype-bounded polymorphism as follows:

```
Fun $sort_3 = (\forall t \leq^{(t)} Person).(\forall c \in^{(c)} t).\lambda x : array\ of\ c.\{$
          $n = size(x);$
          for $i = 1$ to $n - 1$ do
              for $j = 1$ to $n - i$ do {
                  $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1], x[j])$
                  $else\ if\ x[j].younger(x[j+1])\ then$ continue
                  $else\ if\ (x[j+1].name() \leq x[j].name())$
                      $then\ swap(x[j+1], x[j])$
              };
          }
     $sort_3 : (\forall t \leq^{(t)} Person).(\forall c \in^{(c)} t).array\ of\ c \rightarrow array\ of\ c$
```

where type variable t always expects a subtype of *Person*. The difference between $sort_1$ and $sort_3$ is obvious. That is, $sort_1$ is applicable to all the classes of *Person*, and $sort_3$ is applicable to all the subtypes of *Person*. A subtype-bounded function is typed according to the following rule:

$$\frac{A,\ E,\ R \cup \{t \leq^{(t)} \tau\} \models e : \eta}{A,\ E,\ R \models (\forall t \leq^{(t)} \tau).e : (\forall t \leq^{(t)} \tau).\eta} \tag{4.13}$$

where R is a set of subtype restrictions defined in the previous chapter.

A subtype-bounded function can be used as a function from types to type-bounded functions. The one-step application of a subtype-bounded function is typed according to the following rule:

$$\frac{A,\ E,\ R \models e : (t \leq^{(t)} \tau).\eta, \quad R \models \tau' \leq^{(t)} \tau}{A,\ E,\ R \models e(\tau') : \{\tau'/t\}\eta}. \tag{4.14}$$

It is obvious that subtype-bounded polymorphism is a generalized form of type-bounded polymorphism. For example, when $sort_3$ is called with type $Person$, a type-bounded function is formed as follows:

```
Fun $sort_3(Person) = (\forall c \in^{(c)} Person).\lambda x : array\ of\ c.\{$
                  $n = size(x);$
                  for $i = 1$ to $n - 1$ do
                      for $j = 1$ to $n - i$ do {
                          $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1],\ x[j])$
                          $else\ if\ x[j].younger(x[j+1])\ then$ continue
                          $else\ if\ (x[j+1].name() \leq x[j].name())$
                              $then\ swap(x[j+1],\ x[j])$
                      };
                  }
          $sort_3(Person) : (\forall c \in^{(c)} Person).array\ of\ c \rightarrow array\ of\ c$
```

which is the same as $sort_1$ except for its function name. Of course, $sort_3$ can be applied to an array of objects directly. In this case, a type τ and a class ξ also have to be present such that τ is a subtype of $Person$, ξ is a class of τ, and all elements of the array are of ξ.

The description of a subtype-bounded function may be simplified using the following abbreviation:

$$\xi \leq^{(c)}: \tau \longleftrightarrow (\exists \sigma : TOP)(\xi \in^{(c)} \sigma \wedge \sigma \leq^{(t)} \tau).$$

That is, ξ is regarded as a class of type τ if there exists a type σ such that ξ is a class of σ and σ is a subtype of τ. For example, $sort_3$ can be rewritten as follows:

```
Fun $sort'_3 = (\forall c \leq^{(c)}: Person).\lambda x : array\ of\ c.\{$
```

```
        $n = size(x)$;
        for $i = 1$ to $n - 1$ do
            for $j = 1$ to $n - i$ do {
                $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1],\ x[j])$
                $else\ if\ x[j].younger(x[j+1])\ then$ continue
                $else\ if\ (x[j+1].name() \leq x[j].name())$
                    $then\ swap(x[j+1],\ x[j])$
            };
        }
    $sort'_3 : (\forall c \leq^{(c)}: Person).array\ of\ c \rightarrow array\ of\ c.$
```

The one-step application of a simplified subtype-bounded function is typed according to the following rule:

$$\frac{A,\ E,\ R \models e : (c \leq^{(c)}: \tau).\eta,\ \ R \models \tau' \leq^{(t)} \tau,\ \ E \models \xi \in^{(c)} \tau'}{A,\ E,\ R \models e(\xi) : \{\xi/c\}\eta}. \tag{4.15}$$

4.3.3 Remark

Any type is a subtype of itself. Hence, subtype-bounded polymorphism is a generalized form of type-bounded polymorphism. However, inheritance is not subtype. For example, type *Employee* is a heir type of type *Person* rather than a subtype of type *Person*. Hence, $sort_3$ would not be applicable to *Employee* and other heir types of *Person* according to the meaning of "$\forall t \leq^{(t)} Person$".

The reader may notice that the function body of $sort_3$ can be reused for *Employee* and other heir types of *Person*. Hence, we should weaken the restriction imposed on $sort_3$. In this case, we should have a kind that characterizes the commonality of human beings. Such a kind is *Human* as defined in the previous chapter:

```
    Kind $Human$ = [ $name$    : $\mapsto Single(string)$,
                 $younger$ : thiskind $\mapsto Single(bool)$
               ].
```

Obviously, *Person*, *Employee*, and *Student* are all types of kind *Human*. Using this kind, we can define the following kind-bounded function:

```
Fun $sort_4 = (\forall t \in^{(t)} Human).(\forall c \in^{(c)} t).\lambda x : array\ of\ c.\{$
        $n = size(x);$
        for $i = 1$ to $n - 1$ do
            for $j = 1$ to $n - i$ do {
                $if\ x[j+1].younger(x[j])\ then\ swap(x[j+1],\ x[j])$
                $else\ if\ x[j].younger(x[j+1])\ then$ continue
                $else\ if\ (x[j+1].name() \leq x[j].name())$
                    $then\ swap(x[j+1],\ x[j])$
            };
        }
    $sort_4 : (\forall t \in^{(t)} Human).(\forall c \in^{(c)} t).array\ of\ c \rightarrow array\ of\ c$
```

which has the same function body as $sort_3$, and is applicable to all heir types of *Person*. This is another example showing that kind-bounded polymorphism is more powerful than other forms of polymorphism.

As mentioned before, subtype-bounded polymorphism can be treated as a special form of kind-bounded polymorphism since $Power(\tau)$ is a special kind. Hence, it is not necessary to keep subtype-bounded polymorphism as an independent form of polymorphism in the four-level type hierarchy.

4.4 Algorithmic Abstraction

In the previous sections, three forms of polymorphism were introduced to define polymorphic functions at different abstraction levels. In this section, we will clarify the relation between polymorphism and algorithmic abstraction.

4.4.1 Algorithms

An algorithm is a precise and unambiguous specification of a sequence of steps that can be applied to a certain range of types of objects and always terminates. Thus, there are two major parts in an algorithm definition:

(1) a domain of types that are proper for the algorithm, and

(2) a sequence of steps that specify a particular computation approach.

Since the domain of an algorithm usually includes a variety of types, every step of the algorithm has to be polymorphic to all types in the domain. Thus, algorithms are usually defined in the form of pseudo-code, and, as a result, they need to be re-digested and re-coded into functions (procedures) by programmers. We can describe an algorithm in many ways, such as natural languages, flow charts, and *pseudo* programming languages.

4.4.1.1 Natural Languages

Natural languages, such as English, are commonly used in writing algorithms. For example, an algorithm for finding the minimum element in a set of comparable elements can be written as follows:

```
algorithm [Find the minimum in a set of comparable elements]
        (1) Select an element from the set, and write it to $min$.
        (2) Check if the set is empty. If yes, go to (5); continue otherwise.
        (3) Select an element from the set. If this element is less than $min$,
            write it to $min$.
        (4) Go back to (2).
        (5) Print the content of $min$.
```

In this algorithm, we use the English word "less than" instead of "<", "write" instead of ":=", and so on. Hence, it can be easily understood by programmers. However, we must be careful in writing such an algorithm so that each of its steps is unambiguous. In particular, we have to carefully organize such an algorithm so that its control structure is clear.

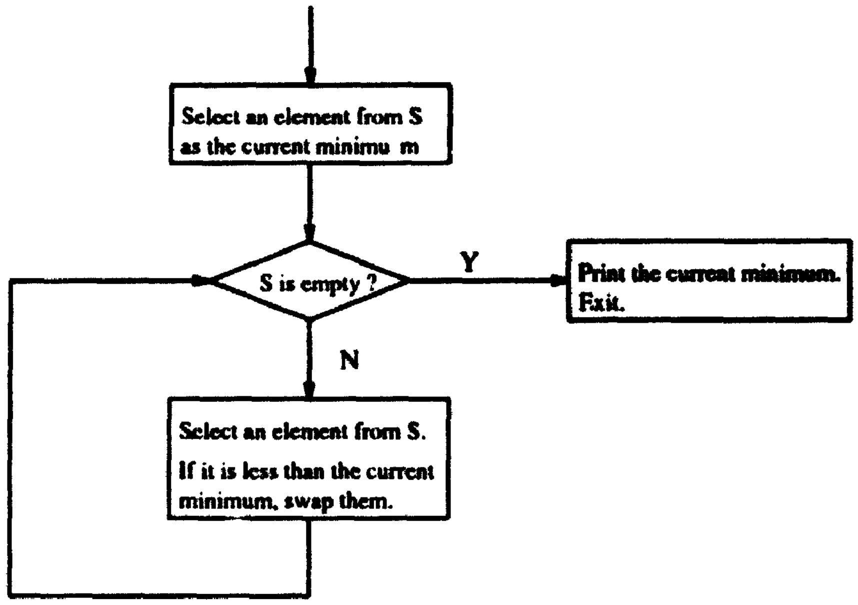
Select an element from S as the current minimu m
S is empty ?
Y
Print the current minimum. Exit.
N
Select an element from S.
If it is less than the current minimum, swap them.

Figure 4.1: A flow chart for finding minimum in a set

4.4.1.2 Flow Charts

Another commonly used way of defining algorithms is to use a flow chart, which is an improved form of a natural language by coupling its use with graphics. For example, the above algorithm can be rewritten in a flow chart which is shown in Figure 4.1.

In this algorithm, the control structure is clearly expressed by arrows and boxes. Hence, the logical structure of an algorithm is shown more clearly. However, each step of such an algorithm is still described using natural languages. Therefore, special attention is still needed in using a flow chart so that each step is unambiguous.

4.4.1.3 *Pseudo* Programming Languages

Algorithms can also be written in pseudo programming languages which contain commonly used control structures and statements. For example, we can write algorithms in a *Pascal*-like language called pseudo-*Pascal* [37]. In pseudo-*Pascal*, the algorithm for finding the minimum in a set of comparable elements can be written as follows:

```
algorithm minimum (var aa: list of comparable elements, n: integer)
    { find the minimum in aa, which contains n integers }
     var i : integer;
     begin
          min := the first element of aa;
          for i := 2 to n
          begin
               temp := the ith element of aa;
               if (temp less than min)
                   then min := temp;
          end;
          print(min);
     end;
```

Obviously, it is very close to a *Pascal* program. Although it still needs to be re-digested by programmers, the effort is dramatically decreased compared to the same algorithm written in a natural language or a flow chart.

4.4.2 Language-Supported Algorithms

An algorithm is usually applicable to infinitely many data types. This makes it impossible for algorithm designers to write a compilable function for each applicable data type. As a result, algorithms have been written in natural languages, flow charts, or pseudo programming languages. An algorithm is essentially an abstraction of a group of functions, which needs to be re-digested and re-coded into functions by programmers. In other words, from an algorithm a concrete function can be obtained for each applicable type under the effort of programmers. We can express the relation between algorithms and functions using the translating function:

$$\textit{translation} : \textbf{algorithm} \rightarrow (\textbf{type} \rightarrow \textbf{function})$$

where **algorithm**, **type**, and **function** denote algorithms, types, and functions, respectively. It means that, for an algorithm and a proper type, the algorithm can be turned to a concrete function which is applicable to all objects of the type.

Unfortunately, in the past, the above translating process is based on the programmers' understanding and re-coding. In general, more complicated algorithms need more programmers' efforts. Hence, in practice, programmers often use those algorithms that are easy to translate but obsolete. In addition, different programmers may produce totally different programs from the same algorithm. Thus, it is not guaranteed that an algorithm is translated into a correct function each time.

Note that, for a given algorithm α, *translation*(α) represents a function from types to functions. If we express the domain of *translation*(α) with **type**(α), then *translation*(α) is a function with the following type structure:

$$translation(\alpha) : \mathbf{type}(\alpha) \rightarrow \mathbf{function}.$$

In general, *translation*(α) is a partial function over the set of types since **type**(α) may not include all types. Obviously, once *translation*(α) is written as a compilable function in a programming language, we can obtain concrete functions just by calling it with proper types. In order to allow algorithm designers to write their algorithms in a compilable form, we have to design programming languages that are endowed with

(1) a type hierarchy to support the definition of algorithm domains, and

(2) constructs to define the step sequences of algorithms which can be instantiated into concrete functions for proper types.

In our framework, a kind defines a set of types that satisfy a set of common properties. In addition, an algorithm often demands only certain properties on each type of its domain and it does not require these properties to be of certain specific types. Also, two or more parameters of an algorithm may be of different types that satisfy the same set of properties. All of these characteristics can be properly defined by kinds. Therefore, kinds are well suited for defining the domains of algorithms. In addition, kind-bounded functions can be used as functions from types to type-bounded functions, or functions from classes to monomorphic functions. Hence, a kind-bounded function essentially represents *translation*(α) for a certain algorithm α. In other words, a kind-bounded function is a language-supported algorithm. For example, the *minimum* algorithm would be written in kind-bounded polymorphism as follows:

```
Fun $minimum = (\forall c \in^{(t)}: TotalOrder).\ \lambda a : array\ of\ c.\{$
                                        $n := size(a);$
                                        $x := a[1];$
                                        for $i := 2$ to $n$ do
                                            $if\ a[i].\leq(x)\ then\ x := a[i];$
                                        return($x$);
                                        }
    $minimum : (\forall c \in^{(t)}: TotalOrder).array\ of\ c \rightarrow c.$
```

Hence, any programming language which is incorporated with kinds and kind-bounded polymorphism meets the requirements for programming languages supporting algorithmic abstraction.

A kind κ has a set of properties that can be used to define the step sequence of an algorithm with κ as its domain. For example, $\{\leq\}$ is the set of properties corresponding to *TotalOrder*. From $\{\leq\}$ we can construct other properties using logical operations, such as "=" and "<", where $x = y$ iff $x \leq y$ and $y \leq x$, and $x < y$ iff $x \leq y$ and $x \neq y$. For any kind κ, we use $\theta(\kappa)$ to denote the set of properties specified in κ, and $power(\theta(\kappa))$ to denote the set of properties that can be derived from $\theta(\kappa)$. Thus, $\theta(TotalOrder) = \{\leq\}$, and $\{\leq, =, <\} \subseteq power(\theta(TotalOrder))$. Obviously, any property in $power(\theta(\kappa))$ can be used to specify an algorithm which has κ as its domain.

The properties in $power(\theta(\kappa))$ can be instantiated by any type in κ. For example, any type in *TotalOrder* has "≤", "=", and "<" as abstract methods although different names may be used for them. If some property in $power(\theta(TotalOrder))$ has not been implemented in a type of *TotalOrder*, it can be defined according to its definition in *TotalOrder*. Since different types of a kind κ may use different names for the properties in $power(\theta(\kappa))$, the instantiation of a κ-bounded algorithm also includes the renaming of some properties.

4.4.3 Remark

Although kind-bounded polymorphism can be used to express algorithms at the programming language level, there are some different points between informally-written algorithms and language-supported algorithms, which are summarized below.

(1) Informally-written algorithms are described using informal languages, such as natural languages and *pseudo* programming languages, but language-supported algorithms are described using precise programming languages. Therefore, extra work is often needed to translate informally-written algorithms to actual programs in programming languages. In contrast, language-supported algorithms are already programs (polymorphic functions) from the user's viewpoint. Therefore, they can be used directly without any translation.

(2) In an informally-written algorithm, it is usually not explicitly specified what is the domain of types for the algorithm. In contrast, the designers of language-supported algorithms have to be precise about the domains of their algorithms. For example, it should be explicitly specified what is the minimum set of properties that are required by an algorithm.

Chapter 5

Algorithmic Abstraction in Kind-C++

In this chapter, we present a concrete programming language, called KC (Kind-C++), which has been designed and implemented to demonstrate the use of the four-level type hierarchy in supporting algorithmic abstraction. KC is an extension of C++ with the notion of *kinds*. Our KC preprocessor, which has been partially implemented [51, 59, 60], accepts KC programs and produces G++[1] programs.

5.1 Problems of C++

In the late eighties, templates were introduced to C++ to define "families of classes and functions" [53, 23]. Since C++ has been designed based on the two-level type structure, i.e., objects and classes, templates actually implement *Form 1* universal-bounded polymorphism. Here, we look at the problems associated with *Form 1* universal-bounded polymorphism from a more practical point of view. We start with a simple example.

Suppose that a template function for the bubble sort algorithm is available in a template library (pp.271, [53])

```
template<class C> void sort( Vector<C>& v)
{      // bubble sort
       int n = v.size();
       for (int i=0; i<n-1; i++)
```

[1]G++ is a GNU C++ compiler

```
                    for (int j=n-1; i<j; j--)
                        if (v[j] <= v[j-1]) {
                            C temp = v[j];
                            v[j] = v[j-1];
                            v[j-1] = temp;    }
            }.
```

A programmer tries to use this template function in his/her program to sort a vector of objects of class X. However, the programmer cannot simply read the heading of the template and then use it since some classes are *odd* to the template (A class is *odd* to a template if the class does not possess the methods required by the template). The programmer has to check whether class X can be used to substitute for the class parameter *C*. In other words, the entire body of the template function has to be read to figure out what methods a valid substituting class for *C* should possess. Reading through the code, the programmer finds that any class that is to replace *C* should have the operator "<=". Then the operator "<=" for class X has to be defined if "<=" has not been defined for X. This process is not difficult for a simple template like the one above, but can be very tedious and error-prone for a complicated template. Several approaches have been suggested to solve the *odd* class problem in [53]; however, these approaches are based on programming methodology rather than on programming language constr s.

There are several inter-related problems raised by this example:

(1) Although C++, which evolved from C, is not a strongly typed language, all variables and formal parameters are required to be properly declared and class conversions are to be explicitly specified. The class requirement for an actual parameter is specified as the class of the cc ponding formal parameter at the header of the function declaration. Apparently, the declarations of class parameters in templates do not follow this current standard of C and other constructs of C++. A c. ss parameter of a template, e.g., *C* in the above example, is totally unrestricted at the header of its declaration. The restrictions

for C do exist, but they are implied in the body of the template rather than explicitly specified. Therefore, templates are not syntactically compatible with other C++ constructs.

(2) In function-based (procedure-based) software reuse, we create reusable code such that users of this code only need to know *what* the code does but not *how* it works. In other words, users should be able to use a reusable subprogram without knowing its implementation details. Object-oriented programming languages are intended to provide constructs and means for software reuse in such a plug-and-play fashion. However, templates do not fit into this category. In order to capture all the necessary properties associated with a class variable in a template, users are normally required to read and understand the entire code of the template. This is just what we try so hard to avoid.

(3) A class parameter of a template is intended to be instantiated by many different actual classes. But, in a template definition, each operator of a parameterized class can be specified only in a fixed notation, which may not satisfy all intended classes. For example, the notation '<=' is valid for the comparison of two integers, floats, or doubles, but it may not be valid for two strings or two objects of another class. Many solutions have been proposed in [53]. However, all of them are awkward, obscure, and difficult to read and write.

In order to solve the above problems, we have designed and implemented the language KC, which supports algorithmic abstraction. The four-level hierarchy of entities is suitable for algorithmic abstraction as we argued. However, only objects and classes are supported in C++. Hence, we need to add structures for types and kinds.

5.2 Abstract Classes and Types

In C++, a class is called an *abstract class* if it possesses one or more pure virtual methods. For example, the following is an abstract class for sequences of integers:

```
class sequence {
      public:
      virtual void insert(int) = 0;
      virtual int  get(void)   = 0;
      virtual int  empty(void) = 0;
}
```

where the keyword **virtual** identifies a virtual method, and a virtual method is made *pure* by an initializer = 0.

Each virtual method of an abstract class can have different versions in different derived classes. Hence, an abstract class essentially represents an interface to a set of classes. We call an abstract class that has only pure virtual methods a *pure* abstract class.

A type can be roughly represented by a pure abstract class. Hence, in KC, a type is treated as a syntactic sugar for a pure abstract class rather than a new entity. In this way, we do not need to introduce extra structures for types. For example, the following is a type for sequences of integers:

```
type sequence {
      // for any x of int, any s of sequence,
      // (s.insert(x)).get = x
      // (s.insert(x)).empty = 0 (false)
      public:
      virtual void insert(int) = 0;
      virtual int  get(void)   = 0;
      virtual int  empty(void) = 0;
}.
```

Our preprocessor will automatically translate it into the corresponding abstract class by simply changing the keyword **type** to the keyword **class**.

A type can have any number of classes, which are implemented with different data structures and different algorithms. Class CC is said to be of type TT if for each virtual method m in **TT** there is a corresponding method m' in **CC** such that the following conditions hold:

(1) m' and m have the same identifier.

(2) m' realizes the behaviour specified by m.

(3) m' has the same input/output types as m.

The behaviour of a virtual method is defined simply by identifiers and comments and understood by programmers' intuition and common sense. Since a type is a syntactic sugar for a pure abstract class, the following is the simplest way to associate a class with its type:

```
class CC : public TT {
      // implementation details
}
```

where **CC** should have a corresponding method for each virtual method in **TT**. For example, we can define a class of type **sequence** using a linked list as follows:

```
class seq1 : public sequence {
      struct node {
             int content;
             char *link;
       };
       node *n;
       public:
             void insert(int) { ---- };
             int  get(void)   { ---- };
             int  empty(void) { ---- };
}.
```

For simplicity, we omit the details of the implementation. We can also define another class of **sequence** using an array as follows:

```
class seq2 : public sequence {
      int n[2404];
      int index;
      public:
            void insert(int) { --- };
            int  get(void)   { --- };
            int  empty(void) { --- };
}.
```

Although **seq1** and **seq2** are different classes, they provide exactly the same service to the outside world. Hence, they are the same from the users' point of view.

A pure virtual method that is not defined in a derived class is still a pure virtual method. Thus, we can define an *heir* type of a given type by adding more pure virtual methods. For example, we can define type **list** by inheriting from **sequence** as follows:

```
type  list : public sequence {
      // for any x of int, any s of list,
      // (s.insert(x)).rest = s
      public:
      virtual list& rest(void) = 0;
}
```

where virtual methods **insert**, **get**, and **empty** are inherited from **sequence**. Similarly, it is also a syntactic sugar for the following class definition:

```
class list : public sequence {
      // for any x of int, any s of list,
      // (s.insert(x)).rest = s
      public:
      virtual list& rest(void) = 0;
}.
```

5.3 Kinds

In *KC*, a kind is an independent entity, which is defined by a group of properties. A property is a function declaration without function body. The following is an example of a kind definition:

```
kind tos {
     // for any x, y, z of some type in tos:
     // x.<=(x)
     // x.<=(y) or y.<=(x)
     // x.<=(y) and y.<=(z) => x.<=(z)
     int operator <= (tos);
}.
```

A class `C` is said to be of kind `tos` if it has a method `int operator <= (C)`. Note that a type is a syntactic sugar for a pure abstract class. Hence, it is not necessary to associate classes with kinds through types.

In KC we do not provide language constructs for defining the semantics of the properties in kinds. Instead, the semantics of these properties are defined simply by identifiers and comments. We do not think this is a weak point of KC since we believe that this is the proper level where formalism should stop for practical programming.

New kinds can be defined from existing kinds using kind inheritance in KC. Thus, we do not need to define every kind from scratch. Its use can be illustrated with the following example:

```
kind semigroup : tos {
     // for any x, y, z of some type in semigroup:
     // x.+(y) = y.+(x),
     // (x.+(y)).+(z) = x.+(y.+(z)),
     // x.*(y.+(z)) = (x.*(y)).+(y.*(z))
     semigroup~ operator +(semigroup~);
     semigroup~ operator *(semigroup~);
}
```

where the symbol "~" is used to indicate that all occurrences of **semigroup** should be matched by the same class. The **semigroup** kind is a *subkind* of **tos**. Obviously, any class of **semigroup** is also a class of **tos**.

In the above example, only one property is inherited by **semigroup** from **tos**. The advantages of kind inheritance will be apparent if many properties need to be copied from a kind to another kind. Multiple kind inheritance is also possible in KC. Consider the following kind definition:

```
kind K1 : K2, K3 {
        //
        int opn();
}
```

which means that kind **K1** will contain the properties of both **K2** and **K3** plus the property **int opn()**.

The syntax for a declaration of a kind is defined as follows:

$$
\begin{array}{lll}
< kind\_specifier > & ::= & < kind\_head > \{:< derivation >\}_0^1 \\
 & & `\{\text{'} < property\_list > `\}\text{'}; \\
< kind\_head > & ::= & < kind\_key > < kind\_name > \\
< kind\_name > & ::= & < identifier > \\
< kind\_key > & ::= & \textbf{kind} \\
< derivation > & ::= & < kind\_name > \mid < derivation >, < kind\_name > \\
< property\_list > & ::= & < property > \mid < property\_list >; < property >
\end{array}
$$

where $\{S\}_0^1$ means that the number of occurrences of S should be 0 or 1. A property is a function definition that has no function body. The syntax of properties can be found in [51]. There are no data members in the declaration of a kind. All properties are public.

Definition 5.1 A class C is said to possess a property p of a kind K if it has a method m such that the following conditions hold:

(1) The name of m is the same as the name of p (This condition will be broadened by the name-aliasing mechanism).

(2) m and p have exactly the same number of parameters.

(3) If the ith parameter in p is a class C', then the ith parameter in m must be C'.

(4) If the ith parameter in p is a kind K', then the ith parameter in m must be a class of K'.

(5) For some i and j, if the ith and jth parameters are of the same kind marked with the same number of "$\sim$" in p, then the ith and jth parameters in m must be the same class of the kind.

If class C possesses all the properties of kind K, then we say that C is a class of K.

In KC, we also provide a name-aliasing mechanism between the methods of a class and the properties of a kind. The following is a renaming declaration using name-aliasing.

```
class C in kind K {
    c1 is k2;
    c2 is k3;
    c3 is k1;
}.
```

It states that C is a class in K under the correspondence that the method c1 of class C corresponds to the property k2 in K, c2 corresponds to k3, and so on. In other words, if class C is to replace a class parameter restricted to K, c1 is to replace k2, c2 is to replace k3, and c3 is to replace k1. A function name or an operator symbol can appear on either side of the "is" statement. For example, for the following class triple

```
class triple {
     int element1;
     int element2;
```

```
          int element3;
     public:
          int leq(triple);
          // ......
     }
```

we can declare the following renaming:

```
     class triple in kind tos{
          leq is operator <=;
     }.
```

5.4 Algorithms over Types

As we have said, all classes of a type share a set of common methods from the users' point of view. Therefore, a function defined for a class C may be generalized to one that is applicable to all classes of the type to which class C belongs. To explicitly express the reusability of such a function, we use the following heading:

```
algorithm <Type T C>
```

which means that class variable `C` ranges over the set of classes defined by type `T`. Such a function is called a *type-bounded* algorithm since it is applicable to all classes defined by a given type. For example, the following is a type-bounded algorithm that calculates the sum of all integers in a list of integers:

```
algorithm <Type list C>
int sum (C& s) {
     int v = 0;
     while ( !(s.empty(void)) ) {
          v += s.get(void);
          s = s.rest(void);
     };
```

```
            return(v);
        }
```

where the keyword **algorithm** denotes an algorithm in KC, and the **Type** clause introduces a type restriction. Only these classes implementing **list** can be used as actual class parameters of the algorithm. Our KC preprocessor will recognize such a type-bounded algorithm, and translate it into a corresponding C++ function. For example, the **sum** algorithm will be transferred into the following C++ function:

```
int sum (list& s) {
        int v = 0;
        while ( !(s.empty(void)) ) {
                v += s.get(void);
                s = s.rest(void);
        };
        return(v);
}.
```

Hence, the **sum** algorithm is a syntactic sugar for the **sum** function. The syntax of calling a type-bounded algorithm is the same as that of calling a function. Thus, any call to the **sum** algorithm is also a call to the **sum** function.

Since a type in KC is a syntactic sugar for a pure abstract class in C++, types may not define the domains of algorithms properly in some situations. Consider the algorithm that selects the smaller object from two comparable objects. We first define a type as follows:

```
type com {
        public:
        virtual int operator <= (com&) = 0;
}.
```

Obviously, from type **com** we can derive many classes that have a total order, such as class C_1 for integers and class C_2 for strings. Then the algorithm can be written in KC as follows:

```
algorithm (Type com C)
int lessthan (C& x, C& y) {
    int v;
    if x.<=(y) then v = 1;
    else v = 0;
    return(v);
}
```

which will be translated into the following function by KC.

```
int lessthan (com& x, com& y) {
    int v;
    if x.<=(y) then v = 1;
    else v = 0;
    return(v);
}
```

Objects of a subclass can be used where objects of a superclass are expected. Hence, objects of class C_1 would be compared with objects of class C_2. However, it is meaningless to compare an integer with a string. Hence, in general, types are not appropriate for defining the domains of algorithms.

5.5 Algorithms over Kinds

A kind defines a set of classes that have common properties. Since the properties of a kind are independent of any class of the kind, any function written only using the properties of a kind is also independent of any class of the kind. In this sense, a function defined over a kind is an algorithm, called a *kind-bounded* algorithm, which is applicable to all classes of the kind. A kind-boundeu algorithm is introduced by using the following heading:

```
algorithm <Kind K C>
```

which means that class variable C ranges over the set of classes defined by kind K.

Now we consider the previously mentioned bubble sort example. It is obvious that only the '<=' operator is associated with concrete classes. Therefore, we can rewrite it into the following kind-bounded algorithm:

```
algorithm <Kind tos C>
void sort( Vector<C>& v) {
    // bubble sort
    int n = v.size();
    for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
            if (v[j] <= v[j-1]) {
                C temp = v[j];
                v[j] = v[j-1];
                v[j-1] = temp;
            }
}.
```

We can easily see that the bubble sort algorithm can be applied to any vector of objects whose class is of kind tos.

Any call to a kind-bounded algorithm will be caught by the KC preprocessor. If the preprocessor finds any incorrect call to a kind-bounded algorithm, it prints out an error message. The syntax of calling a kind-bounded algorithm is the same as the one of calling a function template. Hence, the preprocessor would simply translate the above algorithm into the following template:

```
template <class C>
void sort( Vector<C>& v) {
    // bubble sort
    int n = v.size();
    for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
```

```
                    if (v[j] <= v[j-1]) {
                        C temp = v[j];
                        v[j] = v[j-1];
                        v[j-1] = temp;
                    }
        }.
```

Any incorrect call to **sort** will be caught by the KC preprocessor. Hence, the translation would not cause any problem at run-time.

Based on kind **tos**, the algorithm that selects the smaller object from two comparable objects can be rewritten as follows:

```
algorithm (kind tos C)
int lessthan' (C x, C y) {
    int v;
    if x.<=(y) then v = 1;
    else v = 0;
    return(v);
}
```

Now both x and y are required to be of the same class. Hence, it cannot be used to compare an integer object with a string object. Thus, kinds are more appropriate for defining the domains of algorithms than types.

A kind defines a set of classes in KC. However, not every set of classes can be defined by a kind. A special kind, which consists of all classes, is the universal kind. In KC, the universal kind is denoted by **class**, and any algorithm over **class** is simply defined by a template. For example,

```
template <class C>
void swap(C& x, C& y) {
     C z;
     z = x; x = y; y = z;
}.
```

5.6 Summary

In this chapter, we have given a brief description of the Kind-C++ programming language. The details of its implementation can be found in [51]. Compared with templates defined over the set of all classes, algorithms over types or kinds have the following advantages:

(1) The user of such an algorithm need not read through the body of the algorithm to check for the conditions implicitly imposed by the algorithm body on its class parameters. Now, those conditions are explicitly stated by a type or a kind. Thus, the plug-and-play style of software reuse is supported by KC algorithms.

(2) The type system is stronger and more consistent. Similar to a formal parameter of a function being restricted to a specific class, a class parameter of an algorithm is restricted to a type or a kind. Whether a type or a kind is valid domain of an algorithm can be checked automatically by compiler.

(3) Proving program correctness can become realistic due to the reusability and the higher abstraction level of algorithms, as well as the better-defined interface of algorithms.

Chapter 6

An Algebraic Model of Classes, Types, and Kinds

In the previous chapters, classes, types, and kinds have been defined syntactically and understood by programmer's intuition and common sense. In this chapter, we will give precise definitions of classes, types, and types based on an object-oriented algebraic theory. As we will see, a class corresponds to a ***structured algebra***, a type corresponds to a set of isomorphic structured algebras, and a kind corresponds to a set of structured algebras satisfying the same set of axioms.

6.1 Many-Sorted Signatures and Algebras

Since the early seventies, a prevailing opinion is that a data type is a many-sorted algebra [40, 32, 33]. On the other hand, a many-sorted algebra can be regarded as a model of an axiomatized signature (S, F, M), where S is a set of sorts, F is an $S^* \times S$-sorted set, and M is a set of axioms. Thus, (S, F, M) is often taken as a behavioural specif[illegible]on for a group of data types.

Consider the classes that implement stacks of natural numbers. Obviously, these classes can vary in implementation. They may be implemented with an array, a linked list, or some other structure. Although they are different from each other in their internal representations, they behave to the outside in exactly the same way. In order to specify these classes, we first define the following two signatures which are for natural numbers and boolean values, respectively.

Signature *nat* =
  sort: (*nat*)
  operation:
    zero : $\rightarrow$ (*nat*)
    succ : (*nat*) $\rightarrow$ (*nat*)
Endsignature

Signature *bool* =
  sort: (*bool*)
  operation:
    true : $\rightarrow$ (*bool*)
    false : $\rightarrow$ (*bool*)
Endsignature

Here each signature ς is supposed to have a sort of *interest*, denoted (ς). For example, (*nat*) is a sort of interest in signature *nat*, and (*bool*) is a sort of interest in signature *bool*. Then, the signature for stacks of natural numbers can be defined from *nat* and *bool* by adding more sorts, operations, and axioms as follows:

Signature *stack* = **extend** *nat* and *bool* **with**
  sort: (*stack*)
  operation:
    newstack : $\rightarrow$ (*stack*)
    push : (*stack*) $\times$ (*nat*) $\rightarrow$ (*stack*)
    top : (*stack*) $\rightarrow$ (*nat*)
    pop : (*stack*) $\rightarrow$ (*stack*)
    empty : (*stack*) $\rightarrow$ (*bool*)
  axiom: $\forall s \in (stack), \forall x \in (nat)$
    $top(push(s, x)) = x$,
    $pop(push(s, x)) = s$,
    $empty(newstack) = true$,
    $empty(push(s, x)) = false$.
Endsignature.

Note that we can also define *stack* from scratch by including all associated sorts, operations, and axioms. It can be proven that both ways will generate semantically equivalent axiomatized signatures [57]. Also notice that (*nat*) and (*bool*) denote two sorts. Hence, we have to use (*nat*) and (*bool*) in *stack* rather than *nat* and *bool*.

Any model of an axiomatized signature is a many-sorted algebra, which consists of a set of carrier domains and a set of operations on these domains. For example, a many-sorted algebra of signature *stack* consists of the carrier domains for (*nat*), (*bool*), and (*stack*), as well as the operations for *newstack*, *push*, *top*, *pop*, *empty*, *zero*, *succ*, *true*, and *false*. In general, a carrier domain is a set of structureless elements unless specific operations are associated with them. For example, an array of natural numbers can represent an ordered set, a stack, or even a tree depending on what operations are defined on the array. In this sense, we say that all carrier domains of a many-sorted algebra are in a *flat* structure. Unfortunately, such a flat structure is not appropriate for modelling classes.

In the object-oriented programming environment, a class can be defined with other classes as domains [31, 53, 43]. In other words, any method of a class can have objects of other classes as operands. A class is an integrated unit of data structures local to the class and methods manipulating these data structures, and therefore it can be managed independently of other classes. Thus, we can implement a complicated data type by building it up on simple classes. For instance, in C++ ([53]) *nat* and *bool* would be implemented as two classes that have internal types for (*nat*) and (*bool*), respectively, and *stack* would be implemented as a class that has these two classes as its domains. The internal types corresponding to (*nat*) and (*bool*) are not visible to the *stack* class. In contrast, any many-sorted algebra of signature *stack* has the carrier domains for sorts (*nat*) and (*bool*), respectively. Hence, many-sorted algebras do not fit naturally into the structure of classes.

The disadvantages of many-sorted algebras become more apparent when they are used to model complicated classes. For example, we have a class, named *set_of_stack*, each object of which is a set of objects of the *stack* class. The implementation of the *stack* class is hidden from *set_of_stack*, and therefore its internal type is not directly accessible to *set_of_stack*. Instead, any access to the internal type of the *stack* class has to go through the methods of the *stack* class. However, a many-sorted algebra corresponding to *set_of_stack* includes all the carriers and operations specified in signature *stack*. Indeed, we can specify a complicated data type in a structured way by using the operations over axiomatized signatures [12, 58, 57]. However, such an axiomatized signature still has many-sorted algebras as its models [57]. Therefore, we should have a new algebraic structure in which a carrier for a sort can be not only a set of simple elements but also an algebra. As a result, a complicated many-sorted algebra can be defined by a number of simple structured algebras.

6.2 Structured Signatures and Algebras

In the rest of this chapter, we develop a new form of signatures, called *structured signatures*, to define the semantics of the four-level type hierarchy. A structured signature consists of three parts: a set of *interior sorts*, a set of *exterior sorts*, and a set of *operation symbols* on these sorts. An exterior sort is also a structured signature, and an interior sort is a sort of *interest*. An algebra of a structured signature, called a *structured algebra*, is similar to a many-sorted algebra except that its carrier domains can be other structured algebras. Elements of a structured algebra are called *configurations*.

Multiple interior sorts of a structured signature can be expressed as a tuple of interior sorts. For simplicity, we assume that any structured signature has only one interior sort, which corresponds to an internal type we have defined for classes. For related concepts and definitions, the reader may refer to the following papers on algebraic specifications [30, 32, 33, 40, 57].

6.2.1 Structured Signatures

Structured signatures are based on the notation of sorted sets. Let S be a set of sorts. An S-*sorted* set X is a family of sets X_s, where s ranges over the set S. We denote the S-sorted set X by $\{X_s\}_{s \in S}$. For simplicity, x is said to be an element of X if $x \in X_s$ for some $s \in S$.

Definition 6.1 A *structured signature* Σ is a pair (S, F) such that

(1) $S = \{s_I\} \cup S_E$, denoted $\mathcal{S}(\Sigma)$, where s_I is the sort of interest, denoted $\mathcal{I}(\Sigma)$, and S_E is a set of structured signatures, denoted $\mathcal{E}(\Sigma)$;

(2) F is an $S^* \times S$-sorted family $\{F_{\omega,s}\}_{(\omega,s) \in S^* \times S}$, denoted $\mathcal{F}(\Sigma)$.

The definition is recursive. That is, a structured signature is defined in terms of other structured signatures. Σ is called a *defined* structured signature, and each

$\varsigma \in S_E$ is called a ***base*** structured signature of Σ. If S_E is empty, then Σ is called a ***basic*** structured signature.

Each $F_{\omega,\varsigma} \in F$ is a set of ***operation symbols***. If $f \in F_{\omega,\varsigma}$, then f is said to have ***domain*** ω and ***range*** ς. If ω is $\varsigma_1\varsigma_2\cdots\varsigma_n$, i.e., $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \rightarrow \varsigma$, then we say that f is an n-ary operation symbol.

Note that any operation symbol can be overloaded in a structured signature. That is, there may exist an operation symbol f in F such that $f \in F_{\omega_1,\varsigma_1} \cap F_{\omega_2,\varsigma_2}$, and either $\omega_1 \neq \omega_2$ or $\varsigma_1 \neq \varsigma_2$. The hierarchical structure of a structured signature can be characterized with a ***signature chart*** which is defined below.

Definition 6.2 Let Σ be a structured signature. The ***signature chart*** of Σ, denoted $\mathcal{C}(\Sigma)$, is a directed graph defined as follows:

(1) Σ is a node of $\mathcal{C}(\Sigma)$ (the root node);

(2) If $\mathcal{E}(\Sigma) \neq \emptyset$, for each $\varsigma \in \mathcal{E}(\Sigma)$, let $\mathcal{C}(\varsigma)$ be the signature chart of ς.

Then $\mathcal{C}(\varsigma)$ is a component and $< \Sigma, \varsigma >$ is a directed edge of $\mathcal{C}(\Sigma)$.

The *depth* of $\mathcal{C}(\Sigma)$, denoted $\mathcal{D}(\Sigma)$, is 1 if Σ is basic; otherwise, $\mathcal{D}(\Sigma) = 1 + max\{\mathcal{D}(\varsigma) | \varsigma \in \mathcal{E}(\Sigma)\}$.

Each node in $\mathcal{C}(\Sigma)$ is a structured signature. We use $node(\mathcal{C}(\Sigma))$ to denote the set of all nodes in $\mathcal{C}(\Sigma)$. A structured signature is said to be ***well-defined*** if there is no directed cycle in its signature chart. In the sequel, we assume that all structured signatures are well-defined unless otherwise stated. The following are a few examples of structured signatures.

Example 6.1 A structured signature for natural numbers:

Signature Σ_{nat} =

interior sort: (Σ_{nat})

operation:

$zero : \ \rightarrow (\Sigma_{nat})$

$succ : (\Sigma_{nat}) \rightarrow (\Sigma_{nat})$

Endsignature

Example 6.2 A structured signature for boolean values:

> **Signature** Σ_{bool} =
>
> **interior sort**: (Σ_{bool})
>
> **operation**:
>
> $true \;:\; \rightarrow (\Sigma_{bool})$
>
> $false :\; \rightarrow (\Sigma_{bool})$
>
> **Endsignature**

Σ_{nat} and Σ_{bool} are basic structured signatures since both of them have empty sets of exterior sorts. Obviously, there is no difference between many-sorted signatures and structured signatures for types like *nat* and *bool*.

Example 6.3 A structured signature for stacks of natural numbers:

> **Signature** Σ_{stack} =
>
> **interior sort**: (Σ_{stack})
>
> **exterior sort**: Σ_{bool}, Σ_{nat}
>
> **operation**:
>
> $newstack :\; \rightarrow (\Sigma_{stack})$
>
> $push \;:\; (\Sigma_{stack}) \times \Sigma_{nat} \rightarrow (\Sigma_{stack})$
>
> $top \;:\; (\Sigma_{stack}) \rightarrow \Sigma_{nat}$
>
> $pop \;:\; (\Sigma_{stack}) \rightarrow (\Sigma_{stack})$
>
> $empty \;:\; (\Sigma_{stack}) \rightarrow \Sigma_{bool}$
>
> **Endsignature**

In this example, Σ_{stack} is defined from existing structured signatures Σ_{nat} and Σ_{bool}. However, Σ_{stack} is different from signature *stack* defined at the beginning of this chapter. That is, Σ_{stack} has structured signatures Σ_{nat} and Σ_{bool} as sorts instead of (Σ_{nat}) and (Σ_{bool}).

Example 6.4 A structured signature for the sets that have stacks of natural numbers as elements:

> **Signature** Σ_{sset} =
>
> **interior sort**: (Σ_{sset})
>
> **exterior sort**: Σ_{bool}, Σ_{stack}
>
> **operation**:
>
> $newset$: $\rightarrow (\Sigma_{sset})$
>
> $insert$: $(\Sigma_{sset}) \times \Sigma_{stack} \rightarrow (\Sigma_{sset})$,
>
> $empty$: $(\Sigma_{sset}) \rightarrow \Sigma_{bool}$,
>
> is_elem : $(\Sigma_{sset}) \times \Sigma_{stack} \rightarrow \Sigma_{bool}$
>
> **Endsignature**

Sorts of a structured signature can be other structured signatures. This is a significant departure from traditional signatures. Later we will see that this departure is important for specifying classes in an implementation-independent way. To distinguish from structured signatures, we will call any signature defined in the traditional way a *flat* signature.

Definition 6.3 Let Σ and Σ' be two structured signatures, and $\Sigma = (S, F)$ and $\Sigma' = (S', F')$. Σ is said to be *signature morphic* to Σ' if there exists a 3-tuple $\delta = (\delta_1, \delta_2, \beta)$, where $\delta_1 : S \rightarrow S'$, $\delta_2 : F \rightarrow F'$, and $\beta = \{\delta_\varsigma\}_{\varsigma \in \mathcal{E}(\Sigma)}$, such that

(1) $\delta_2(f) \in F'_{\delta_1(\omega), \delta_1(\varsigma)}$ for any $f \in F_{\omega,\varsigma}$, where $\omega = \varsigma_1\varsigma_2 \cdots \varsigma_n$ and $\delta_1(\omega) = \delta_1(\varsigma_1)\delta_1(\varsigma_2) \cdots \delta_1(\varsigma_n)$;

(2) ς is signature morphic to $\delta_1(\varsigma)$ under δ_ς, for any $\varsigma \in \mathcal{E}(\Sigma)$.

δ is called a *signature morphism* from Σ to Σ'. δ is said to be *injective* if δ_1 and δ_2 are injective maps satisfying (1) and δ_ς is injective for each $\varsigma \in \mathcal{E}(\Sigma)$. Σ is said to be *signature isomorphic* to Σ' if both δ_1 and δ_2 are bijective maps satisfying (1) and the "signature morphic" condition in (2) is changed to "signature isomorphic".

For simplicity, we assume that all signature morphisms are injective. Let δ be a signature morphism from Σ to Σ'. Σ' is said to be a ***horizontal extension*** of Σ if

$\delta_1(\varsigma) = \varsigma$ for each $\varsigma \in \mathcal{S}(\Sigma)$, and $\delta_2(f) = f$ for each $f \in \mathcal{F}(\Sigma)$. In other words, horizontal extension allows us to define new structured signatures from existing ones by adding new exterior signatures and operation symbols. Hence, it is a kind of inheritance over signatures.

6.2.2 Structured Algebras

All symbols appearing in a structured signature are uninterpreted. Their meanings are given by *structured algebras*. Hence, a structured signature is only a syntactic specification for a set of structured algebras.

Definition 6.4 Let $\Sigma = (S, F)$ be a structured signature. A ***structured algebra A*** of Σ consists of:

(1) an interior domain $A_{\mathcal{I}(\Sigma)}$ for $\mathcal{I}(\Sigma)$, denoted $\mathcal{I}(A)$;

(2) an $\mathcal{E}(\Sigma)$-sorted set of structured algebras $\{A_\varsigma\}_{\varsigma\in\mathcal{E}(\Sigma)}$ if $\mathcal{E}(\Sigma) \neq \emptyset$, where A_ς is a structured algebra of ς;

(3) an n-ary operation $f^A : o(A_{\varsigma_1}) \times o(A_{\varsigma_2}) \times \cdots \times o(A_{\varsigma_n}) \rightarrow o(A_{\varsigma_0})$, abbreviated $f^A : o(A_{\varsigma_1\times\varsigma_2\times\cdots\times\varsigma_n}) \rightarrow o(A_{\varsigma_0})$, for each $f : \varsigma_1\times\varsigma_2\times\cdots\times\varsigma_n \rightarrow \varsigma_0 \in F$, where $o(A_{\varsigma_i})$, $0 \leq i \leq n$, is defined as follows:

(a) if ς is $\mathcal{I}(\Sigma)$, $o(A_{\mathcal{I}(\Sigma)}) = A_{\mathcal{I}(\Sigma)}$;

(b) if $\varsigma_i \in \mathcal{E}(\Sigma)$, $o(A_{\varsigma_i}) = \{(a, \mathcal{F}(\varsigma_i)^{A_{\varsigma_i}}) \mid a \in \mathcal{I}(A_{\varsigma_i})\}$, where $\mathcal{F}(\varsigma_i)^{A_{\varsigma_i}} = \{f^{A_{\varsigma_i}} \mid f \in \mathcal{F}(\varsigma_i)\}$.

For simplicity, we often call A a *Σ-algebra*. Note that $o(A)$ ($= \{(a, \mathcal{F}(\Sigma)^A) | a \in \mathcal{I}(A)\}$) is different from A. Each element $\bar{a}$ in $o(A)$ is called a ***configuration*** of A. If $\bar{a} = (a, \mathcal{F}(\Sigma)^A)$, then a is called the *data* portion of $\bar{a}$, denoted ***data***$(\bar{a})$, and $\mathcal{F}(\Sigma)^A$ the *operation* portion of $\bar{a}$, denoted *opn*$(\bar{a})$. In the rest of this chapter, if it is not stated explicitly, we will use A to denote $o(A)$ for simplicity.

A configuration of a structured algebra is different from an element of a simple set. First of all, it carries all operations of the algebra plus a specific value of the interior domain of the algebra. Two configurations of a structured algebra differ from each other if only if they carry different values of the interior domain of the algebra.

If a structured algebra is used to model a class in an OOPL, then a configuration of the structured algebra corresponds to an internal state carried by an object of the class.

The above definition is recursive. In particular, A is called a ***defined*** structured algebra, and A_ς is called a *base* structured algebra of A for each $\varsigma \in \mathcal{E}(\Sigma)$. We use $Alg(\Sigma)$ to denote the set of all Σ-algebras.

The operations of a defined structured algebra may depend on the operations of its base structured algebras. This is supported by allowing a configuration of a structured algebra to carry all operations of the algeb~a. Let C be a configuration which has an operation f. Then $C.f$ denotes f in C. The following are a few examples of structured algebras.

Example 6.5 A structured algebra A of Σ_{bool}:

> **Algebra** A **of** Σ_{bool} =
>
> > **interior domain:**
> >
> > $A_{(\Sigma_{bool})} =_{def} \{T.\ F\}$
> >
> > **operation:**
> >
> > $true^A =_{def} T$
> >
> > $false^A =_{def} F$
>
> **Endalgebra**

$(T,\ \{true^A, false^A\})$ and $(F,\ \{true^A, false^A\})$ are two examples of configurations of A.

Example 6.6 A structured algebra B of Σ_{nat}:

> **Algebra** B **of** Σ_{nat} =
>
> > **interior domain:**
> >
> > $B_{(\Sigma_{nat})} =_{def} \{0\} \cup \{x + 1 | x \in B_{(\Sigma_{nat})}\}$
> >
> > **operation:**
> >
> > $zero^B =_{def} 0$

$succ^B(a) =_{def} a + 1$

Endalgebra

$(0, \{zero^B, succ^B\})$, $(1, \{zero^B, succ^B\})$, and $(2, \{zero^B, succ^B\})$ are all configurations of B.

Example 6.7 A structured algebra C of Σ_{stack}:

Algebra C **of** Σ_{stack} =

interior domain:

$C_{(\Sigma_{stack})} =_{def} \{\emptyset\} \cup \{[a_1 a_2 \cdots a_n] \mid a_i \in B,\ 1 \leq i \leq n\}$

exterior domain

$C_{\Sigma_{bool}} =_{def} A$

$C_{\Sigma_{nat}} =_{def} B$

operation:

$newstack^C =_{def} \emptyset.$

$push^C(s, x) =_{def} [x] \cdot s,$

$empty^C(s) =_{def}$ if $\|s\| = 0$ then T else F.

$top^C([x] \cdot s) = x,$

$pop^C([x] \cdot s) = s.$

Endalgebra

Here, we use $[\cdots]$ to denote a list; "$s_1 \cdot s_2$" is the list which is obtained by appending s_2 to the tail of s_1, $\emptyset$ is the empty list, and $\|s\|$ is the size of list s. The following is an example of configurations of C.

$([(0, \{zero^B, succ^B\})(2, \{zero^B, succ^B\})], \{newstack^C, top^C, pop^C, push^C, empty^C\})$

Definition 6.5 Let Σ be a structured signature and A a Σ-algebra. The *algebra chart* of A, denoted $\mathcal{C}(A)$, is a directed graph defined as follows:

(1) A is a node in $\mathcal{C}(A)$ (the root node);

(2) For each base algebra A_ς of A, let $\mathcal{C}(A_\varsigma)$ be the algebra chart of A_ς. Then $\mathcal{C}(A_\varsigma)$ is a component and $< A, A_\varsigma >$ is a directed edge of $\mathcal{C}(A)$.

We use $node(\mathcal{C}(A))$ to denote the set of all nodes in $\mathcal{C}(A)$. Obviously, $\mathcal{C}(A)$ is isomorphic to $\mathcal{C}(\Sigma)$ in the sense that $< A_{\varsigma'}, A_{\varsigma''} >\in \mathcal{C}(A)$ iff $< \varsigma', \varsigma'' >\in \mathcal{C}(\Sigma)$. The following theorem states that each structured algebra has only one structured signature up to signature isomorphism.

Theorem 6.1 *For any two structured signatures Σ and Σ', if Σ is signature isomorphic to Σ', then any Σ-algebra is also a Σ'-algebra.*

Proof. Let $\Sigma = (\{s_I\} \cup S_E, F)$, $\Sigma' = (\{s'_I\} \cup S'_E, F')$, and δ be an signature isomorphism between Σ and Σ'. We proceed by induction on the depth $\mathcal{D}(\Sigma)$ of $\mathcal{C}(\Sigma)$.

If Σ is basic, then $S_E = \emptyset$ and $S'_E = \emptyset$. Let A be a Σ-algebra. By the definition of Σ-algebra, for s_I there exists a corresponding domain A_{s_I} in A, and for each operation symbol $f \in F_{\omega,\varsigma}$ there exists a corresponding operation $f^A : A_\omega \to A_\varsigma$ in A, where $\omega \in \{s_I\}^*$ and $\varsigma \in \{s_I\}$. Note that A_{s_I} is the only domain of A, and $\delta_1(s_I) = s'_I$. Hence, we choose A_{s_I} as the carrier domain of s'_I, i.e., $A_{s_I} = A_{s'_I}$. In addition, we choose f^A as an operation for $f' \in F'_{\delta_1(\omega),\delta_1(\varsigma)}$, where $\delta_2(f) = f'$. Thus, A is also a Σ'-algebra.

Suppose that any Σ-algebra is a Σ'-algebra when $\mathcal{D}(\Sigma) = \mathcal{D}(\Sigma') < n$.

Consider the case that $\mathcal{D}(\Sigma) = n$. Let A be a Σ-algebra. Then there are a carrier domain A_{s_I} for s_I and a structured algebra A_ς of depth $< n$ for each $\varsigma \in S_E$. By induction hypothesis, A_ς is also a structured algebra of ς', where $\varsigma' \in S'_E$ and $\delta_1(\varsigma) = \varsigma'$. Hence, we choose A_ς as the algebra of ς' and A_{s_I} as the carrier domain of s'_I.

For each $f' : \varsigma'_1 \times \varsigma'_2 \times \cdots \times \varsigma'_m \to \varsigma'_0 \in F'$, there exists $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_m \to \varsigma_0 \in F$ such that $\delta_2(f) = f'$ and $\delta_1(\varsigma_i) = \varsigma'_i$, $0 \leq i \leq m$. $f^A : A_{\varsigma_1} \times A_{\varsigma_2} \times \cdots \times A_{\varsigma_m} \to A_{\varsigma_0}$ is an operation for f and A_{ς_i} is a carrier domain for ς'_i, $0 \leq i \leq m$. Hence, f^A is also an operation for f', i.e., $f^A = f'^A$. Thus, the proof is complete. □

For traditionally defined signatures Σ and Σ', if Σ is morphic to Σ', then any Σ'-algebra can be reduced into a Σ-algebra [57]. This is not true in structured signatures since two different structured algebras may have different structures for their configurations. However, we have a result for structured signatures which is similar to one for traditionally defined signatures.

Definition 6.6 Let $\delta = (\delta_1, \delta_2, \{\delta_\varsigma\}_{\varsigma \in \mathcal{E}(\Sigma)})$ be a signature morphism from Σ to Σ' and A' a Σ'-algebra. The *coercion* of A' with respect to δ, denoted c_Σ, is a family $\{c_\varsigma | \varsigma \in \mathcal{S}(\Sigma)\}$, where

(a) if Σ is basic, $c_{\mathcal{I}(\Sigma)}$ is an identity function on $A'_{\mathcal{I}(\Sigma')}$,

(b) if Σ is not basic, c_ς is the coercion of A'_ς with respect to δ_ς for each $\varsigma \in \mathcal{E}(\Sigma)$ and $c_{\mathcal{I}(\Sigma)}$ is a function from $A'_{\mathcal{I}(\Sigma')}$ to $c_{\mathcal{I}(\Sigma)}(A'_{\mathcal{I}(\Sigma')})$ $(= \{c_{\mathcal{I}(\Sigma)}(a) | a \in A'_{\mathcal{I}(\Sigma')}\})$

such that

(1) $c_\varsigma(\delta_2(f)^{A'}(a_1, a_2, \cdots, a_n)) = \delta_2(f)^{A'}(c_{\varsigma_1}(a_1), c_{\varsigma_2}(a_2), \cdots, c_{\varsigma_n}(a_n))$ for each $\delta_2(f) : \delta_1(\varsigma_1) \times \delta_1(\varsigma_2) \times \cdots \times \delta_1(\varsigma_n) \to \delta_1(\varsigma) \in \mathcal{F}(\Sigma')$, $a_i \in A'_{\delta_1(\varsigma_i)}$ $(1 \leq i \leq n)$;

(2) $c_\Sigma(a) = (c_{\mathcal{I}(\Sigma)}(data(a)), \delta_2(\mathcal{F}(\Sigma))^{A'})$ for each $a \in A'$, where $\delta_2(\mathcal{F}(\Sigma))^{A'} = \{\delta_2(f)^{A'} | f \in \mathcal{F}(\Sigma)\}$.

Intuitively, c_Σ is a function which translates configurations of a Σ'-algebra into ones of a Σ-algebra.

Definition 6.7 Let $\delta = (\delta_1, \delta_2, \{\delta_\varsigma\}_{\varsigma \in \mathcal{E}(\Sigma)})$ be a signature morphism from Σ to Σ', A' a Σ'-algebra, and c_Σ the coercion of A' with respect to δ. The ***δ-reduction*** of A' with respect to c_Σ, denoted $A'|_{\delta, c_\Sigma}$, is defined as follows:

(1) $(A'|_{\delta, c_\Sigma})_{\mathcal{I}(\Sigma)} = c_{\mathcal{I}(\Sigma)}(A'_{\mathcal{I}(\Sigma')})$;

(2) $(A'|_{\delta, c_\Sigma})_\varsigma = A'_{\delta_1(\varsigma)}|_{\delta_\varsigma, c_\varsigma}$ if $\varsigma \in \mathcal{E}(\Sigma)$, where δ_ς is a signature morphism from ς to $\delta_1(\varsigma)$, and c_ς is the coercion of $A'_{\delta_1(\varsigma)}$ with respect to δ_ς;

(3) for any $\delta_2(f) : \delta_1(\varsigma_1) \times \delta_1(\varsigma_2) \times \cdots \times \delta_1(\varsigma_n) \to \delta_1(\varsigma) \in \mathcal{F}(\Sigma')$ and each $a_i \in A'_{\delta_1(\varsigma_i)}$ $(1 \leq i \leq n)$, if $c_{\varsigma_i}(a_i) = b_i$, $f^{A'|_{\delta, c_\Sigma}}(b_1, b_2, \cdots, b_n) = c_\varsigma(\delta_2(f)^{A'}(a_1, a_2, \cdots, a_n))$.

Theorem 6.2 *Let δ be a signature morphism from Σ to Σ', and A' be a Σ'-algebra. Then $A'|_{\delta, c_\Sigma}$ is a Σ-algebra.*

Proof. It is obvious from the construction of $A'|_{\delta, c_\Sigma}$. □

Now we define morphisms between structured algebras.

Definition 6.8 Let $\Sigma = (S, F)$ be a structured signature. For any two Σ-algebras A and B, an *algebra morphism* from A to B is $\{\psi^\varsigma : A_\varsigma \to B_\varsigma \mid \varsigma \in S\}$, denoted ψ^Σ, such that

(1) ψ^ς is a mapping from A_ς to B_ς if $\varsigma = \mathcal{I}(\Sigma)$,

(2) ψ^ς is an algebra morphism from A_ς to B_ς for each $\varsigma \in \mathcal{E}(\Sigma)$ if $\mathcal{E}(\Sigma) \neq \emptyset$, and

(3) $\psi^\varsigma(f^A(a_1, a_2, \cdots, a_n)) = f^B(\psi^{\varsigma_1}(a_1), \psi^{\varsigma_2}(a_2), \cdots, \psi^{\varsigma_n}(a_n))$
for each $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \to \varsigma \in F$.

If $a \in A$, then $\psi^\Sigma(a) = (\psi^{\mathcal{I}(\Sigma)}(data(a)), \mathcal{F}(\Sigma)^B)$.

ψ^Σ is called a *surjective* (*injective*) algebra morphism from A to B if $\psi^{\mathcal{I}(\Sigma)}$ is a surjective (injective) map and ψ^ς is a surjective (injective) algebra morphism for each $\varsigma \in \mathcal{E}(\Sigma)$. ψ^Σ is called an *algebra isomorphism* between A and B if $\psi^{\mathcal{I}(\Sigma)}$ is a bijective map and ψ^ς is an algebra isomorphism for each $\varsigma \in \mathcal{E}(\Sigma)$. If ς is clear in context, $\varsigma \in \mathcal{S}(\Sigma)$, we simply use ψ^Σ to denote ψ^ς.

A is called an *initial* in $Alg(\Sigma)$ if for any Σ-algebra B there is a unique algebra morphism from A to B. Σ-algebra A is called a *terminal* in $Alg(\Sigma)$ if for any Σ-algebra B there is a unique algebra morphism from B to A.

6.3 Axiomatized Structured Signatures

A structured signature is only a syntactic specification for a set of structured algebras. To specify structured algebras that satisfy some properties, we need to append axioms to a structured signature. Axioms can be defined in different ways [57]. Here, we are only interested in such structured algebras that satisfy some equations [30, 29].

6.3.1 Σ-Terms and Σ-Equations

Each structured signature defines a set of expressions, called *terms*, which are formed from variables and operation symbols. Terms are used to construct equations which are needed to specify the behaviours of structured algebras.

Definition 6.9 Let $\Sigma = (S, F)$ be a structured signature, and X_Σ be a set of variables. The set of *ς-terms* for each $\varsigma \in S$, denoted $T(\Sigma, X_\Sigma)_\varsigma$, contains:

(1) each $x \in X_\Sigma$ and each nullary operation symbol $f : \to \varsigma \in F$ if $\varsigma = \mathcal{I}(\Sigma)$;

(2) each $(t, \mathcal{F}(\varsigma))$ if $\varsigma \in \mathcal{E}(\Sigma)$, where $t \in T(\varsigma, X_\varsigma)_{\mathcal{I}(\varsigma)}$ and X_ς is a set of variables;

(3) each $f(t_1, t_2, \cdots, t_n)$, where $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \to \varsigma$ is an operation symbol in F, and $t_i \in T(\Sigma, X_\Sigma)_{\varsigma_i}$, $1 \leq i \leq n$;

(5) all those generated using the above rules in a finite number of steps.

We use $\mathcal{X}(\Sigma)$ to denote $\{X_\varsigma | \varsigma \in node(\mathcal{C}(\Sigma))\}$. For simplicity, we assume that $X_\varsigma \cap X_{\varsigma'} = \emptyset$ for any ς, $\varsigma' \in node(\mathcal{C}(\Sigma))$. Each element in $\{(t, \mathcal{F}(\Sigma)) | t \in T(\Sigma, X_\Sigma)_{\mathcal{I}(\Sigma)}\}$ is called a *Σ-term*. In particular, $(x, \mathcal{F}(\Sigma))$ is called a Σ-term *variable*, where $x \in X_\Sigma$. We will often use $\bar{x}$ to denote $(x, \mathcal{F}(\Sigma))$.

Definition 6.10 Let $\Sigma = (S, F)$ be a structured signature. The *term algebra* of Σ, denoted $T(\Sigma, X_\Sigma)$, has

(1) $T(\Sigma, X_\Sigma)_{\mathcal{I}(\Sigma)}$ as the carrier domain of $\mathcal{I}(\Sigma)$, and

(2) an $\mathcal{E}(\Sigma)$-sorted set of term algebras $\{T(\varsigma, X_\varsigma)\}_{\varsigma \in \mathcal{E}(\Sigma)}$ if $\mathcal{E}(\Sigma) \neq \emptyset$

such that $f^{T(\Sigma, X_\Sigma)}(t_1, t_2, \cdots, t_n) = f(t_1, t_2, \cdots, t_n)$ for each $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \to \varsigma \in F$, where $t_i \in o(T(\varsigma_i, X_{\varsigma_i}))$ if $\varsigma_i \in \mathcal{E}(\Sigma)$, and $t_i \in T(\Sigma, X_\Sigma)_{\mathcal{I}(\Sigma)}$ if $\varsigma_i = \mathcal{I}(\Sigma)$, $1 \leq i \leq n$.

If each variable set in $\mathcal{X}(\Sigma)$ is empty, then $T(\Sigma, X_\Sigma)$ is called the *word algebra* of Σ, denoted $T(\Sigma, \emptyset)$. Each element in $T(\Sigma, \emptyset)$ is called a Σ-word.

Let Σ ' ? a structured signature and A be a Σ-algebra. The family of maps $\{v_\varsigma : X_\varsigma \to \mathcal{I}(A_\varsigma) | \varsigma \in node(\mathcal{C}(\Sigma)),\ X_\varsigma \in \mathcal{X}(\Sigma),\ and\ A_\varsigma \in node(\mathcal{C}(A))\}$, abbreviated as $v : \mathcal{X}(\Sigma) \to A$, is called a *value assignment* of $\mathcal{X}(\Sigma)$ in A. The *interpretation* of Σ-terms in A with respect to v is $I^{(v)} = \{I^{(v)}_\varsigma | \varsigma \in \mathcal{S}(\Sigma)\}$, where $I^{(v)}_\varsigma$ is a map from $T(\Sigma, X_\Sigma)_\varsigma$ to A_ς if $\varsigma = \mathcal{I}(\Sigma)$ and $I^{(v)}_\varsigma$ is the interpretation of ς-terms in A_ς with respect to v if $\varsigma \in \mathcal{E}(\Sigma)$, such that

(1) $I^{(v)}_{\mathcal{I}(\Sigma)}(x) = v_\Sigma(x)$ for $x \in X_\Sigma$, and

(2) $I^{(v)}_\varsigma(f(t_1, t_2, \cdots, t_n)) = f^A(I^{(v)}_{\varsigma_1}(t_1), I^{(v)}_{\varsigma_2}(t_2), \cdots, I^{(v)}_{\varsigma_n}(t_n))$

for $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \to \varsigma \in \mathcal{F}(\Sigma)$ and $t_i \in T(\Sigma, X_\Sigma)_{\varsigma_i}$, $1 \leq i \leq n$,

where $I^{(v)}_{\varsigma_i}(t_i) = (I^{(v)}_{\mathcal{I}(\varsigma_i)}(data(t'_i)), \mathcal{F}(\varsigma_i)^{A_{\varsigma_i}})$ if $t_i = (t'_i, \mathcal{F}(\varsigma_i))$.

It is obvious that $I^{(v)}$ is an algebra morphism from $T(\Sigma, X_\Sigma)$ to A. In particular, the interpretation of Σ-words in A, denoted I, is independent of any value assignment.

Definition 6.11 Let Σ be a structured signature. A Σ-*equation* e is $t_1 =_\varsigma t_2$, where t_1 and t_2 are terms of sort $\varsigma \in \mathcal{S}(\Sigma)$. Let A be a Σ-algebra and v be a value assignment from $\mathcal{X}(\Sigma)$ to A. A is said to *satisfy* e with respect to v, denoted $A \models_{\{\Sigma,v\}} e$, if $I^{(v)}_\varsigma(t_1) = I^{(v)}_\varsigma(t_2)$.

If $A \models_{\{\Sigma,v\}} e$ holds for any value assignment v, then A is said to *satisfy* e, denoted $A \models_\Sigma e$. Let M be a set of Σ-equations. If $A \models_\Sigma e$ holds for each e in M, then A is said to *satisfy* M, denoted $A \models_\Sigma M$.

Theorem 6.3 *Let ψ^Σ be an algebra isomorphism between Σ-algebras A and B. Then for each Σ-equation e, $A \models_\Sigma e$ iff $B \models_\Sigma e$.*

Proof. Let e be $t_1 =_\varsigma t_2$ and $A \models_\Sigma e$. For any value assignment $u : \mathcal{X}(\Sigma) \rightarrow B$, we construct the value assignment $v : \mathcal{X}(\Sigma) \rightarrow A$ such that $v(x) = a$ iff $u(x) = b$ and $\psi^\Sigma(a) = b$. Obviously, $A \models_{\{\Sigma,v\}} e$, i.e., $I^{(v)}_\varsigma(t_1) = I^{(v)}_\varsigma(t_2)$. Now we prove that $B \models_{\{\Sigma,u\}} e$.

If $\varsigma = \mathcal{I}(\Sigma)$, $I^{(v)}_\varsigma(t_1)$ and $I^{(v)}_\varsigma(t_2)$ are elements in $\mathcal{I}(A)$. Hence, $\psi^\Sigma(I^{(v)}_\varsigma(t_1)) \in \mathcal{I}(B)$, $\psi^\Sigma(I^{(v)}_\varsigma(t_2)) \in \mathcal{I}(B)$, and $\psi^\Sigma(I^{(v)}_\varsigma(t_1)) = \psi^\Sigma(I^{(v)}_\varsigma(t_2))$.

If $\varsigma \in \mathcal{E}(\Sigma)$, then $I^{(v)}_\varsigma(t_1)$ and $I^{(v)}_\varsigma(t_2)$ are configurations of A_ς. Hence, both $\psi^\Sigma(I^{(v)}_\varsigma(t_1))$ and $\psi^\Sigma(I^{(v)}_\varsigma(t_2))$ are configurations of B_ς, and $\psi^\Sigma(I^{(v)}_\varsigma(t_1)) = \psi^\Sigma(I^{(v)}_\varsigma(t_2))$.

Note that $\psi^\Sigma(I^{(v)}_\varsigma(t_1)) = I^{(u)}_\varsigma(t_1)$ and $\psi^\Sigma(I^{(v)}_\varsigma(t_2)) = I^{(u)}_\varsigma(t_2)$. Hence, $I^{(u)}_\varsigma(t_1) =_\varsigma I^{(u)}_\varsigma(t_2)$, i.e., $B \models_{\{\Sigma,u\}} e$. In addition, u is any value assignment. Thus, $B \models_\Sigma e$.

Similarly, $B \models_\Sigma e$ implies $A \models_\Sigma e$. Thus we finish the proof. □

6.3.2 Axiomatized Structured Signatures

For any structured signature Σ, an axiomatized structured signature corresponding to Σ is obtained by associating each structured signature in $node(\mathcal{C}(\Sigma))$ with a set of axioms.

Definition 6.12 Let $\Sigma = (S, F)$ be a structured signature, where $S = \{s_I\} \cup S_E$. An *axiomatized structured signature* $\hat{\Sigma}$ corresponding to Σ is a 3-tuple $(\hat{S}, \hat{F}, \hat{M})$, where

(1) if Σ is a basic structured signature, then $\hat{S} = S$, $\hat{F} = F$, and $\hat{M}$ is a set of Σ-equations;

(2) if Σ is a non-basic structured signature, then

(a) $\hat{S} = \{\hat{s}_I\} \cup \{\hat{\varsigma} | \varsigma \in S_E\}$, where $\hat{\varsigma}$ is an axiomatized structured signature corresponding to ς for each $\varsigma \in S_E$,

(b) $\hat{F} = \{\hat{f} : \hat{\varsigma}_1 \times \hat{\varsigma}_2 \times \cdots \times \hat{\varsigma}_n \to \hat{\varsigma} | f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \to \varsigma \in F\}$, and

(c) $\hat{M} = \{\hat{e} | e \in M\}$, where M is a set of Σ-equations and $\hat{e}$ is obtained from e by replacing each sort ς with $\hat{\varsigma}$ and each operation symbol f with $\hat{f}$.

Similarly, we can define the signature chart of $\hat{\Sigma}$. For any axiomatized structured signature $\hat{\Sigma} = (\hat{S}, \hat{F}, \hat{M})$, we use $\mathcal{F}(\hat{\Sigma})$, $\mathcal{S}(\hat{\Sigma})$, and $\mathcal{M}(\hat{\Sigma})$ to denote $\hat{S}$, $\hat{F}$, and $\hat{M}$, respectively. Obviously, $\hat{\Sigma}$ is signature isomorphic to Σ if we ignore $\mathcal{M}(\hat{\varsigma})$ for each $\varsigma \in node(\mathcal{C}(\hat{\Sigma}))$. For simplicity, in the sequel we still use Σ to denote $\hat{\Sigma}$. We use *AxSign* to denote the set of all axiomatized structured signatures. For any $\Sigma \in AxSign$, we use $\Gamma(\Sigma)$ to denote the structured signature which is obtained from Σ by removing $\mathcal{M}(\varsigma)$ for each $\varsigma \in node(\mathcal{C}(\Sigma))$.

Example 6.8 An axiomatized structured signature for stacks of natural numbers:

Signature Σ_{stack} =

interior sort: (Σ_{stack})

exterior sort: Σ_{bool}, Σ_{nat}

operation:

newstack : $\to (\Sigma_{stack})$,

push : $(\Sigma_{stack}) \times \Sigma_{nat} \to (\Sigma_{stack})$,

top : $(\Sigma_{stack}) \to \Sigma_{nat}$

pop : $(\Sigma_{stack}) \to (\Sigma_{stack})$

empty : $(\Sigma_{stack}) \to \Sigma_{bool}$,

axiom:

$\forall s \in (\Sigma_{stack}), \bar{x} \in \Sigma_{nat}$,

$top(push(s, \bar{x})) = \bar{x}$,

$pop(push(s, \bar{x})) = s$,

$empty(newstack) = true$,

$empty(push(s, \bar{x})) = false$.

Endsignature

Definition 6.13 Let Σ be an axiomatized structured signature. A $\Gamma(\Sigma)$-algebra A is said to be a *Σ-algebra* if $A_{\Gamma(\varsigma)} \models_{\Gamma(\varsigma)} \mathcal{M}(\varsigma)$ for each $A_{\Gamma(\varsigma)} \in node(\mathcal{C}(A))$, where $\varsigma \in node(\mathcal{C}(\Sigma))$, and $A_{\Gamma(\Sigma)} = A$.

A Σ-algebra is also a $\Gamma(\Sigma)$-algebra. But, a $\Gamma(\Sigma)$-algebra is not necessarily a Σ-algebra. Thus, Σ is not only a syntactic specification but also a behaviour specification for a set of structured algebras. Although the sorts and operation symbols of Σ are still uninterpreted, the axioms of Σ do impose restrictions on ''s structured algebras. We still use $Alg(\Sigma)$ to denote the set of all Σ-algebras.

Definition 6.14 Let $\Sigma = (S, F, M)$ and $\Sigma' = (S', F', M')$ be two axiomatized structured signatures. Σ is said to be *axiomatized-signature morphic* to Σ' if there exists a 4-tuple $\delta = (\delta_1, \delta_2, \delta_3, \beta)$, where $\delta_1 : S \rightarrow S'$, $\delta_2 : F \rightarrow F'$, $\delta_3 : M \rightarrow M'$, and $\beta = \{\delta_\varsigma\}_{\varsigma \in \mathcal{E}(\Sigma)}$, such that

(1) $\delta_2(f) \in F'_{\delta_1(\omega), \delta_1(\varsigma)}$ for each $f \in F_{\omega,\varsigma}$, where $\omega = \varsigma_1 \cdots \varsigma_n$ and $\delta_1(\omega) = \delta_1(\varsigma_1) \cdots \delta_1(\varsigma_n)$;

(2) For each $e : t_1 =_\varsigma t_2 \in M$, there is $\delta_3(e) \in M'$, where $\delta_3(e) : \delta_2(t_1) =_{\delta_1(\varsigma)} \delta_2(t_2)$ and $\delta_2(t)$ is defined as follows:

(a) if $t = x$, then $\delta_2(x) = x$,

(b) if $t = (a, \mathcal{F}(\varsigma))$, then $\delta_2((a, \mathcal{F}(\varsigma))) = (\delta_2(a), \mathcal{F}(\delta_1(\varsigma)))$,

(c) if $t = f(t_1, t_2, \cdots, t_n)$, $\delta_2(f(t_1, t_2, \cdots, t_n)) = \delta_2(f)(\delta_2(t_1), \delta_2(t_2), \cdots, \delta_2(t_n))$;

(3) ς is axiomatized-signature morphic to $\delta_1(\varsigma)$ under δ_ς, for each $\varsigma \in \mathcal{E}(\Sigma)$.

Here, for simplicity, we assume that $T(\Sigma, X_\Sigma)$ and $T(\Sigma', X_{\Sigma'})$ use the same variables. We call the 4-tuple δ an *axiomatized-signature morphism* from Σ to Σ'. We call

δ a *subjective (injective)* axiomatized-signature morphism if δ_1, δ_2, and δ_3 are surjective (injective) and δ_ς is a surjective (injective) axiomatized-signature morphism for each $\varsigma \in \mathcal{E}(\Sigma)$. We call δ an *axiomatized-signature isomorphism* between Σ and Σ' if δ_1, δ_2, and δ_3 are bijective, and δ_ς is an axiomatized-signature isomorphism for each $\varsigma \in \mathcal{E}(\Sigma)$.

Obviously, if Σ is axiomatized-structure morphic to Σ', then $\Gamma(\Sigma)$ is signature morphic to $\Gamma(\Sigma')$. For simplicity, we assume that all axiomatized-signature morphisms are injective.

Definition 6.15 Let $\delta = (\delta_1, \delta_2, \delta_3, \{\delta_\varsigma\}_{\varsigma \in \mathcal{E}(\Sigma)})$ be an axiomatized-signature morphism from Σ to Σ'. The *morphism chart* of δ, denoted $\mathcal{C}(\delta)$, is a directed graph defined as follows:

(1) δ is a node in $\mathcal{C}(\delta)$ (the root node);

(2) For each axiomatized-signature morphism δ_ς from ς to $\delta_1(\varsigma)$, $\varsigma \in \mathcal{E}(\Sigma)$, let $\mathcal{C}(\delta_\varsigma)$ be the morphism chart of δ_ς. Then $\mathcal{C}(\delta_\varsigma)$ is a component and $< \delta, \delta_\varsigma >$ is a directed edge of $\mathcal{C}(\delta)$.

An axiomatized-signature morphism δ from Σ to Σ' can be reduced into a signature morphism from $\Gamma(\Sigma)$ to $\Gamma(\Sigma')$, denoted $\Gamma(\delta)$, by ignoring δ'_3 for each $\delta' \in \mathcal{C}(\delta)$. In addition, a Σ-algebra is also a $\Gamma(\Sigma)$-algebra. Hence, any Σ'-algebra A' can be coerced into a $\Gamma(\Sigma)$-algebra under the coercion $c_{\Gamma(\Sigma)}$ of A' with respect to $\Gamma(\delta)$, denoted $A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}$. The following theorem says that $A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}$ is also a Σ-algebra.

Theorem 6.4 *Let $\delta = (\delta_1, \delta_2, \delta_3, \{\delta_\varsigma\}_{\varsigma \in \mathcal{E}(\Sigma)})$ be an axiomatized-signature morphism from Σ to Σ', and A' be a Σ'-algebra. Then $A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}$ is a Σ-algebra.*

Proof. We need to prove that

$$(A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}})_{\Gamma(\varsigma)} \models_{\Gamma(\varsigma)} \mathcal{M}(\varsigma) \text{ for each } (A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}})_{\Gamma(\varsigma)} \in node(\mathcal{C}(A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}))$$

where $\varsigma \in node(\mathcal{C}(\Sigma))$ and $(A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}})_{\Gamma(\Sigma)} = A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}$.

We only consider the case when $\varsigma = \Sigma$. Other cases can be proven similarly.

Since A' is a Σ'-algebra, we have $A' \models_{\Gamma(\Sigma')} \delta_3(e)$ for each $e \in \mathcal{M}(\Sigma)$. If e is $t_1 = t_2$, then $\delta_3(e)$ is $\delta_2(t_1) = \delta_2(t_2)$. Hence, $A' \models_{\Gamma(\Sigma')} \delta_2(t_1) = \delta_2(t_2)$. For any value assignment $v : \mathcal{X}(\Sigma) \rightarrow A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}$, we define $v' : \mathcal{X}(\Sigma') \rightarrow A'$ such that $v'(x) = a'$ if $v(x) = a$ and $c_\Sigma(a') = a$. Obviously, $A' \models_{\{\Gamma(\Sigma'), v'\}} \delta_2(t_1) = \delta_2(t_2)$, i.e., $I^{(v')}(\delta_2(t_1)) = I^{(v')}(\delta_2(t_2))$. Now we prove inductively that $c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(t_1))) = I^{(v)}(t_1)$.

(i) Let $t_1 = x$. Then

$$\begin{aligned}
c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(x))) &= c_{\Gamma(\Sigma)}(I^{(v')}(x)) \\
&= c_{\Gamma(\Sigma)}(a') = a \\
&= I^{(v)}(x) \\
&= I^{(v)}(t_1).
\end{aligned}$$

(ii) Let $t_1 = f(t_{11}, t_{12}, \cdots, t_{1n})$, and $c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(t_{1i}))) = I^{(v)}(t_{1i})$, $1 \leq i \leq n$. Then

$$\begin{aligned}
& c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(f(t_{11}, t_{12}, \cdots, t_{1n})))) \\
&= c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(f)(\delta_2(t_{11}), \delta_2(t_{12}), \cdots, \delta_2(t_{1n})))) \\
&= c_{\Gamma(\Sigma)}(\delta_2(f)^{A'}(I^{(v')}(\delta_2(t_{11})), I^{(v')}(\delta_2(t_{12})), \cdots, I^{(v')}(\delta_2(t_{1n})))) \\
&= f^{A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}}(c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(t_{11}))), c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(t_{12}))), \cdots, c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(t_{1n})))) \\
&= f^{A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}}(I^{(v)}(t_{11}), I^{(v)}(t_{12}), \cdots, I^{(v)}(t_{1n})) \\
&= I^{(v)}(f(t_{11}, t_{12}, \cdots, t_{1n})).
\end{aligned}$$

Thus, $c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(t_1))) = I^{(v)}(t_1)$. Similarly, we can prove that $c_{\Gamma(\Sigma)}(I^{(v')}(\delta_2(t_2))) = I^{(v)}(t_2)$. Notice that $I^{(v')}(\delta_2(t_1)) = I^{(v')}(\delta_2(t_2))$. Thus, $I^{(v)}(t_1) = I^{(v)}(t_2)$, which implies that $A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}} \models_{\{\Gamma(\Sigma), v\}} t_1 = t_2$. Therefore, $A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}} \models_{\Gamma(\Sigma)} t_1 = t_2$. □

6.3.3 Σ-Classes and Σ-Types

In this subsection, we establish an algebraic model of classes and types based on axiomatized structured signatures. Our definition for types is essentially a generalization of Wirsing's framework for abstract data types [57]. We also assume that each structured algebra satisfies the *principle of generation* [4]. According to this principle, each domain of a structured algebra can only contain such elements that can be generated finitely.

Lemma 6.1 *Let Σ be an axiomatized structured signature. A Σ-algebra A is finitely generated if the interpretation $I : T(\Sigma, \emptyset) \rightarrow A$ is surjective.*

Proof. We proceed by induction on the depth of Σ.

Let Σ be a basic axiomatized structured signature. Then A is a basic structured algebra. If $I : T(\Sigma, \emptyset) \rightarrow A$ is surjective, then for each element a in $A_{\mathcal{I}(\Sigma)}$ there exists an element t in $T(\Sigma, \emptyset)_{\mathcal{I}(\Sigma)}$ such that $I_{\mathcal{I}(\Sigma)}(t) = a$. Note that t is formed finitely from operation symbols. In addition, I is an algebra morphism from $T(\Sigma, \emptyset)$ to A. Therefore, a is a finitely generated element in A.

Suppose that Σ-algebra A is finitely generated if $\mathcal{D}(\Sigma) < n$ and $I : T(\Sigma, \emptyset) \rightarrow A$ is surjective.

Consider Σ of depth n. By induction, A_ς is finitely generated for any $\varsigma \in \mathcal{E}(\Sigma)$. In addition, for each $a \in A_{\mathcal{I}(\Sigma)}$, there exists a $t \in T(\Sigma, X_\Sigma)_{\mathcal{I}(\Sigma)}$ such that a is an image of t under $I_{\mathcal{I}(\Sigma)}$. Therefore, a can be finitely generated from elements in A_ς, $\varsigma \in \mathcal{S}(\Sigma)$. □

Definition 6.16 Let Σ be an axiomatized structured signature and A be a Σ-algebra. A is said to be a Σ-*class* if $I : T(\Sigma, \emptyset) \rightarrow A$ is surjective.

According to Lemma 6.1, any Σ-class is a finitely generated structure. We use $Class(\Sigma)$ to denote the set of all Σ-classes. Σ-class A is called an *initial* of $Class(\Sigma)$ if for any Σ-class B in $Class(\Sigma)$ there exists a unique algebra morphism from A to B. Σ-class A is called an *terminal* of $Class(\Sigma)$ if for any Σ-class B in $Class(\Sigma)$ there exists a unique algebra morphism from B to A. Σ is said to be a *class signature* if $Class(\Sigma)$ is not empty. We use $ClassSign$ to denote the set of all class signatures.

For each class signature Σ, we define a binary relation $\Re$ in $Class(\Sigma)$ such that $A \Re B$ for any $A, B \in Class(\Sigma)$ if there exists an algebra isomorphism between A and B. The relation $\Re$ divides $Class(\Sigma)$ into a number of collections of Σ-classes such that each collection contains only isomorphic Σ-classes. We say that each collection of isomorphic Σ-classes forms a Σ-*type*. In particular, all initial Σ-classes form the *initial* Σ-type, and all terminal Σ-classes form the *terminal* Σ-type. We use $Type(\Sigma)$ to denote the set of all Σ-types.

A Σ-type abstracts away from the concrete representations of its Σ-classes, and, therefore, it is independent of any Σ-class. Σ-types can be defined in terms of *congruences* on $T(\Sigma, \emptyset)$.

Definition 6.17 Let Σ be an axiomatized structured signature and A be a Σ-class. A Σ-*congruence* $\equiv$ on A is $\{\equiv_\varsigma\}_{\varsigma \in \mathcal{S}(\Sigma)}$, where

(1) $\equiv_{\mathcal{I}(\Sigma)}$ is an equivalence relation on $A_{\mathcal{I}(\Sigma)}$;

(2) $\equiv_\varsigma$ is a ς-congruence on A_ς for each $\varsigma \in \mathcal{E}(\Sigma)$;

(3) if $a_i \equiv_{\varsigma_i} b_i$ for $a_i, b_i \in A_{\varsigma_i}$, $1 \leq i \leq n$, then $f^A(a_1, a_2, \cdots, a_n) \equiv_\varsigma f^A(b_1, b_2, \cdots, b_n)$ for any $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \rightarrow \varsigma \in \mathcal{F}(\Sigma)$.

If $a \equiv_{\mathcal{I}(\Sigma)} b$ for any $a, b \in A_{\mathcal{I}(\Sigma)}$, then $(a, \mathcal{F}(\Sigma)^A) \equiv_\Sigma (b, \mathcal{F}(\Sigma)^A)$.

Let Σ be an axiomatized structured signature and A be a Σ-class. A congruence on $T(\Sigma, \emptyset)$ associated with A, denoted $\equiv^A$, can be defined as follows:

(1) For any $t_1, t_2 \in T(\Sigma, \emptyset)_{\mathcal{I}(\Sigma)}$, $t_1 \equiv^A_{\mathcal{I}(\Sigma)} t_2$ iff $I(t_1) = I(t_2)$, where I is the interpretation of Σ-words on A;

(2) $\equiv^A_\varsigma$ is $\equiv^{A_\varsigma}$ for each $\varsigma \in \mathcal{E}(\Sigma)$, where $\equiv^{A_\varsigma}$ is a congruence on $T(\varsigma, \emptyset)$ associated with A_ς.

If $t_1 \equiv^A_{\mathcal{I}(\Sigma)} t_2$, then $(t_1, \mathcal{F}(\Sigma)) \equiv^A (t_2, \mathcal{F}(\Sigma))$.

Note that $\equiv^{A_{\varsigma_i}}_{\mathcal{I}(\varsigma_i)}$ is an equivalence relation on $T(\varsigma_i, \emptyset)_{\mathcal{I}(\varsigma_i)}$ for each $\varsigma_i \in node(\mathcal{C}(\Sigma))$, where $A_{\varsigma_i} = A$ if $\varsigma_i = \Sigma$. Hence, $\equiv^{A_{\varsigma_i}}$ is an equivalence relation on $T(\varsigma_i, \emptyset)$ for each $\varsigma_i \in node(\mathcal{C}(\Sigma))$. We use $[a]$ to express the equivalent class that includes a. The *quotient* of $T(\Sigma, \emptyset)$ with respect to $\equiv^A$, denoted $T(\Sigma, \emptyset)/\equiv^A$, is a $\Gamma(\Sigma)$-algebra which is defined as follows:

(1) $(T(\Sigma, \emptyset)/\equiv^A)_{\mathcal{I}(\Sigma)} =_{def} T(\Sigma, \emptyset)_{\mathcal{I}(\Sigma)}/\equiv^A_{\mathcal{I}(\Sigma)}$;

(2) $(T(\Sigma, \emptyset)/\equiv^A)_\varsigma =_{def} T(\varsigma, \emptyset)/\equiv^{A_\varsigma}$ for each $\varsigma \in \mathcal{E}(\Sigma)$;

(3) $f^{T(\Sigma,\emptyset)/\equiv^A}([t_1], [t_2], \cdots, [t_n]) =_{def} [f(t_1, t_2, \cdots, t_n)]$ for each $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \rightarrow \varsigma \in \mathcal{F}(\Sigma)$, where $[t_i] \in T(\varsigma_i, \emptyset)/\equiv^{A_{\varsigma_i}}$ if $\varsigma_i \in \mathcal{E}(\Sigma)$,

otherwise $[t_i] \in T(\Sigma,\emptyset)/\equiv^A_{\mathcal{I}(\Sigma)}$, $1 \le i \le n$.

If $[t] \in T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}/\equiv^A_{\mathcal{I}(\Sigma)}$, then $[(t,\mathcal{F}(\Sigma))] \in T(\Sigma,\emptyset)/\equiv^A$. If $t_1 \equiv^A_{\mathcal{I}(\Sigma)} t_2$, then $[(t_1,\mathcal{F}(\Sigma))] = [(t_2,\mathcal{F}(\Sigma))]$.

Lemma 6.2 *Let Σ be an axiomatized structured signature. If A is a Σ-class, then A is algebra isomorphic to $T(\Sigma,\emptyset)/\equiv^A$.*

Proof. We proceed inductively on the depth of $\mathcal{C}(\Sigma)$.

Step 1. Let Σ be basic. Then A is also a basic Σ-class. We define $\psi^{\mathcal{I}(\Sigma)} : A_{\mathcal{I}(\Sigma)} \to T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}/\equiv^A_{\mathcal{I}(\Sigma)}$ such that $\psi^{\mathcal{I}(\Sigma)}(a) = [t]$ if $I_{\mathcal{I}(\Sigma)}(t) = a$, where $a \in A_{\mathcal{I}(\Sigma)}$, $t \in T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}$, and I is the interpretation of Σ-words with respect to A. According to the definition of Σ-class, $I_{\mathcal{I}(\Sigma)}$ is a surjective map from $T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}$ to $A_{\mathcal{I}(\Sigma)}$. Therefore, $\psi^{\mathcal{I}(\Sigma)}$ is a well-defined map.

(1) For any $[t] \in T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}/\equiv^A_{\mathcal{I}(\Sigma)}$, there exists an element $a \in A_{\mathcal{I}(\Sigma)}$ such that $I_{\mathcal{I}(\Sigma)}(t) = a$. Thus, $\psi^{\mathcal{I}(\Sigma)}(a) = [t]$. Therefore, $\psi^{\mathcal{I}(\Sigma)}$ is a surjective map from $A_{\mathcal{I}(\Sigma)}$ to $T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}/\equiv^A_{\mathcal{I}(\Sigma)}$.

(2) For any a_1, $a_2 \in A_{\mathcal{I}(\Sigma)}$, If $a_1 \ne a_2$, then $[t_1] \ne [t_2]$, where $\psi^{\mathcal{I}(\Sigma)}(a_1) = [t_1]$ and $\psi^{\mathcal{I}(\Sigma)}(a_2) = [t_2]$. If $[t_1] = [t_2]$, then $I_{\mathcal{I}(\Sigma)}(t_1) = I_{\mathcal{I}(\Sigma)}(t_2)$, which implies that $a_1 = a_2$. This is a contradiction. Therefore, $\psi^{\mathcal{I}(\Sigma)}$ is injective.

(3) By the definition of I, we have the following equation

$$\begin{aligned} I_{\mathcal{I}(\Sigma)}(f(t_1,t_2,\cdots,t_n)) &= f^A(I_{\mathcal{I}(\Sigma)}(t_1), I_{\mathcal{I}(\Sigma)}(t_2),\cdots,I_{\mathcal{I}(\Sigma)}(t_n)) \\ &= f^A(a_1,a_2,\cdots,a_n) \end{aligned}$$

where $I_{\mathcal{I}(\Sigma)}(t_i) = a_i$, $1 \le i \le n$. Thus,

$$\psi^{\mathcal{I}(\Sigma)}(f^A(a_1,a_2,\cdots,a_n)) = [f(t_1,t_2,\cdots,t_n)] = f^{T(\Sigma,\emptyset)/\equiv^A}([t_1],[t_2],\cdots,[t_n]).$$

Therefore, $\psi = \{\psi^{\mathcal{I}(\Sigma)}\}$ is an algebra isomorphism from A to $T(\Sigma,\emptyset)/\equiv^A$.

Step 2. Suppose that A is algebra isomorphic to $T(\Sigma,\emptyset)/\equiv^A$ for any Σ of depth $< n$.

Step 3. Now we consider Σ of depth n. By induction, there exists an algebra isomorphism ψ^ς from A_ς to $T(\varsigma,\emptyset)/\equiv^{A_\varsigma}$ for each $\varsigma \in \mathcal{E}(\Sigma)$ since $\mathcal{D}(\varsigma) < n$. We define a map $\psi^{\mathcal{I}(\Sigma)}$ from $A_{\mathcal{I}(\Sigma)}$ to $T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}/\equiv^{A}_{\mathcal{I}(\Sigma)}$ such that $\psi^{\mathcal{I}(\Sigma)}(a) = [t]$ if $I_{\mathcal{I}(\Sigma)}(t) = a$, where $a \in A_{\mathcal{I}(\Sigma)}$, $t \in T(\Sigma,\emptyset)_{\mathcal{I}(\Sigma)}$, and I is the interpretation of Σ-words with respect to A. Obviously, $\psi^{\mathcal{I}(\Sigma)}$ is bijective. In addition, $\psi^{\varsigma}(f^{A}(a_1,a_2,\cdots,a_m)) = f^{T(\Sigma,\emptyset)/\equiv^{A}_{\varsigma}}([t_1],[t_2],\cdots,[t_m])$, where $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_m \rightarrow \varsigma \in \mathcal{F}(\Sigma)$, $I_{\varsigma_i}(t_i) = a_i$, $a_i \in A_{\varsigma_i}$, $1 \leq i \leq m$. Therefore, $\psi = \{\psi^{\varsigma}\}_{\varsigma\in\mathcal{S}(\Sigma)}$ is an algebra isomorphism from A to $T(\Sigma,\emptyset)/\equiv^{A}$. □

Theorem 6.5 *Let Σ be an axiomatized structured signature. If A is a Σ-class, then $T(\Sigma,\emptyset)/\equiv^{A}$ is also a Σ-class.*

Proof. By Lemma 6.2, there exists an algebra isomorphism between A and $T(\Sigma,\emptyset)/\equiv^{A}$. Thus, $T(\Sigma,\emptyset)/\equiv^{A}$ is a Σ-algebra by Theorem 6.3.

In addition, we can define a surjective algebra morphism h from $T(\Sigma,\emptyset)$ to $T(\Sigma,\emptyset)/\equiv^{A}$ so that $h(t) = [t]$ for any $t \in T(\Sigma,\emptyset)$. Therefore, $T(\Sigma,\emptyset)/\equiv^{A}$ is a Σ-class. □

Lemma 6.3 *Let Σ be an axiomatized structured signature, and A and B be two Σ-classes. If $\equiv^{A} = \equiv^{B}$, then A is algebra isomorphic to B.*

Proof. We define $h^{\Sigma} = \{h^{\varsigma} : A_{\varsigma} \rightarrow B_{\varsigma} | \varsigma \in \mathcal{S}(\Sigma)\}$ as follows:

- for any $a \in A_{\mathcal{I}(\Sigma)}$ and $b \in B_{\mathcal{I}(\Sigma)}$, if there exists a term t such that $I_{\mathcal{I}(\Sigma)}(t) = a$ and $I'_{\mathcal{I}(\Sigma)}(t) = b$, then $h^{\mathcal{I}(\Sigma)}(a) = b$, where I and I' are the interpretations of Σ-words in A and B, respectively;
- for any $\varsigma \in \mathcal{E}(\Sigma)$, h^{ς} is defined as h^{Σ}.

Now we prove that h^{Σ} is an algebra isomorphism between A and B. We proceed inductively on the depth of Σ.

Step 1. Let Σ be basic. Then $h^{\Sigma} = \{h^{\mathcal{I}(\Sigma)} : A_{\mathcal{I}(\Sigma)} \rightarrow B_{\mathcal{I}(\Sigma)}\}$.

(1) Since A is a Σ-class, for each $a \in A_{\mathcal{I}(\Sigma)}$ there exists a Σ-word t such that $I_{\mathcal{I}(\Sigma)}(t) = a$. In addition, $\equiv^{A} = \equiv^{B}$ implies that there exists $b \in B$ such that $I'_{\mathcal{I}(\Sigma)}(t) = b$. Thus, b is the image of a under $h^{\mathcal{I}(\Sigma)}$. Hence, $h^{\mathcal{I}(\Sigma)}$ is a well-defined map.

(2) For any $b \in B_{\mathcal{I}(\Sigma)}$, there exists a word t such that $I'_{\mathcal{I}(\Sigma)}(t) = b$. Since $\equiv^A = \equiv^B$, there exists $a \in A_{\mathcal{I}(\Sigma)}$ such that $I_{\mathcal{I}(\Sigma)}(t) = a$. Hence, $h^{\mathcal{I}(\Sigma)}$ is also surjective.

(3) For any $a_1, a_2 \in A_{\mathcal{I}(\Sigma)}$, there exist t_1 and t_2 in $T(\Sigma, \emptyset)$ such that $I_{\mathcal{I}(\Sigma)}(t_1) = a_1$ and $I_{\mathcal{I}(\Sigma)}(t_2) = a_2$. Since $\equiv^A = \equiv^B$, there exist b_1 and b_2 in $B_{\mathcal{I}(\Sigma)}$ such that $I'_{\mathcal{I}(\Sigma)}(t_1) = b_1$ and $I'_{\mathcal{I}(\Sigma)}(t_2) = b_2$. Thus, $h^{\mathcal{I}(\Sigma)}(a_1) = b_1$ and $h^{\mathcal{I}(\Sigma)}(a_2) = b_2$. Obviously, $b_1 \neq b_2$ if $a_1 \neq a_2$. Hence, $h^{\mathcal{I}(\Sigma)}$ is an injective map.

(4) Let $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \rightarrow \varsigma \in \mathcal{F}(\Sigma)$ and $a_i \in A_{\mathcal{I}(\Sigma)}$, $1 \leq i \leq n$. Then there exists t_i such that $I_{\mathcal{I}(\Sigma)}(t_i) = a_i$, $1 \leq i \leq n$, and $f^A(a_1, a_2, \cdots, a_n) = I_{\mathcal{I}(\Sigma)}(f(t_1, t_2, \cdots, t_n))$ (by the definition of I). Let $I'_{\mathcal{I}(\Sigma)}(f(t_1, t_2, \cdots, t_n)) = f^B(b_1, b_2, \cdots, b_n)$, where $I'_{\mathcal{I}(\Sigma)}(t_i) = b_i$, $1 \leq i \leq n$. According to the definition of h^Σ, $h^{\mathcal{I}(\Sigma)}(f^A(a_1, a_2, \cdots, a_n)) = f^B(b_1, b_2, \cdots, b_n)$, and $h^{\mathcal{I}(\Sigma)}(a_i) = b_i$, $1 \leq i \leq n$. Thus,

$$h^{\mathcal{I}(\Sigma)}(f^A(a_1, a_2, \cdots, a_n)) = f^B(h^{\mathcal{I}(\Sigma)}(a_1), h^{\mathcal{I}(\Sigma)}(a_2), \cdots, h^{\mathcal{I}(\Sigma)}(a_n)).$$

Hence, $h^\Sigma = \{h^{\mathcal{I}(\Sigma)}\}$ is an algebra isomorphism from A to B.

Step 2. Suppose that h^Σ is an algebra isomorphism between A and B if $\mathcal{D}(\Sigma) < n$.

Step 3. Now consider Σ of depth n. By induction, h^ς is an algebra isomorphism between A_ς and B_ς for each $\varsigma \in \mathcal{E}(\Sigma)$. Hence, for any $a \in A_\varsigma$ there exists $b \in B_\varsigma$ such that $h^\varsigma(a) = b$, where $\varsigma \in \mathcal{E}(\Sigma)$.

Obviously, $h^{\mathcal{I}(\Sigma)}$ is also an bijective map between $A_{\mathcal{I}(\Sigma)}$ and $B_{\mathcal{I}(\Sigma)}$.

Let $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_m \rightarrow \varsigma \in \mathcal{F}(\Sigma)$ and $a_i \in A_{\varsigma_i}$, $1 \leq i \leq m$. Then there exists $t_i \in T(\Sigma, \emptyset)_{\varsigma_i}$ such that $I_{\varsigma_i}(t_i) = a_i$, $1 \leq i \leq m$, and $f^A(a_1, a_2, \cdots, a_m) = I_\varsigma(f(t_1, t_2, \cdots, t_m))$. Let $I'_\varsigma(f(t_1, t_2, \cdots, t_m)) = f^B(b_1, b_2, \cdots, b_m)$, where $I'_{\varsigma_i}(t_i) = b_i$, $1 \leq i \leq m$. According to the definition of h^ς, $h^\varsigma(f^A(a_1, a_2, \cdots, a_m)) = f^B(b_1, b_2, \cdots, b_m)$, and $h^{\varsigma_i}(a_i) = b_i$, $1 \leq i \leq m$. Thus,

$$h^\varsigma(f^A(a_1, a_2, \cdots, a_m)) = f^B(h^{\varsigma_1}(a_1), h^{\varsigma_2}(a_2), \cdots, h^{\varsigma_m}(a_m)).$$

Therefore, $h^\Sigma = \{h^\varsigma : A_\varsigma \rightarrow B_\varsigma | \varsigma \in \mathcal{S}(\Sigma)\}$ is an algebra isomorphism between A and B. □

Lemma 6.4 *Let Σ be an axiomatized structured signature, and A and B be two Σ-classes. If A is algebra isomorphic to B, then $\equiv^A = \equiv^B$.*

Proof. Let $\psi^{\Sigma} = \{\psi^{\varsigma} : A_{\varsigma} \rightarrow B_{\varsigma} | \varsigma \in \mathcal{S}(\Sigma)\}$ be an algebra isomorphism between A and B. Then

$$\psi^{\varsigma}(f^A(a_1, a_2, \cdots, a_n)) = f^B(\psi^{\varsigma_1}(a_1), \psi^{\varsigma_2}(a_2), \cdots, \psi^{\varsigma_n}(a_n))$$

for any $f : \varsigma_1 \times \varsigma_2 \times \cdots \times \varsigma_n \rightarrow \varsigma \in \mathcal{F}(\Sigma)$ and $a_i \in A_{\varsigma_i}$, where $\varsigma, \varsigma_i \in \mathcal{S}(\Sigma)$, $1 \leq i \leq n$. We first prove that $\psi^{\varsigma}(I_{\varsigma}(t)) = I'_{\varsigma}(t)$ for any $t \in T(\Sigma, \emptyset)_{\varsigma}$, where I and I' are the interpretations of Σ-terms on A and B, respectively.

If t is a constant function symbol c, then $\psi^{\varsigma}(I_{\varsigma}(c)) = \psi^{\varsigma}(c^A) = c^B = I'_{\varsigma}(c)$.

Suppose that $t = f(t_1, t_2, \cdots, t_n)$ and $\psi^{\Sigma}(I_{\varsigma_i}(t_i)) = I'_{\varsigma_i}(t_i)$, $1 \leq i \leq n$. Thus, we have

$$\begin{aligned}
&\psi^{\varsigma}(I_{\varsigma}(f(t_1, t_2, \cdots, t_n))) \\
&= \psi^{\varsigma}(f^A(I_{\varsigma_1}(t_1), I_{\varsigma_2}(t_2), \cdots, I_{\varsigma_n}(t_n))) \\
&= f^B(\psi^{\varsigma_1}(I_{\varsigma_1}(t_1)), \psi^{\varsigma_2}(I_{\varsigma_2}(t_2)), \cdots, \psi^{\varsigma_n}(I_{\varsigma_n}(t_n))) \qquad (I_{\varsigma_i}(t_i) \in A_{\varsigma_i}) \\
&= f^B(I'_{\varsigma_1}(t_1), I'_{\varsigma_2}(t_2), \cdots, I'_{\varsigma_n}(t_n)) \\
&= I'_{\varsigma}(f(t_1, t_2, \cdots, t_n)).
\end{aligned}$$

Therefore, $\psi^{\varsigma}(I_{\varsigma}(t)) = I'_{\varsigma}(t)$ for any $t \in T(\Sigma, \emptyset)_{\varsigma}$.

For any $t_1, t_2 \in T(\Sigma, \emptyset)_{\varsigma}$, if $t_1 \equiv^A_{\varsigma} t_2$, then $I_{\varsigma}(t_1) = I_{\varsigma}(t_2)$. By the above fact, $\psi^{\varsigma}(I_{\varsigma}(t_1)) = I'_{\varsigma}(t_1)$ and $\psi^{\varsigma}(I_{\varsigma}(t_1)) = I'_{\varsigma}(t_2)$. Hence, $I'_{\varsigma}(t_1) = I'_{\varsigma}(t_2)$. Thus, $\equiv^A \subseteq \equiv^B$.

Similarly, we can prove $\equiv^B \subseteq \equiv^A$. Therefore, $\equiv^A = \equiv^B$. □

Theorem 6.6 *Let A and B be two Σ-class. A is algebra isomorphic to B iff $\equiv^A = \equiv^B$.*

Proof. It is a consequence of Lemma 6.3 and Lemma 6.4. □

This theorem states that all isomorphic Σ-classes corresponds to the same congruence on $T(\Sigma, \emptyset)$. A Σ-type is a set of isomorphic Σ-classes, and therefore it can be identified with a congruence on $T(\Sigma, \emptyset)$.

6.3.4 Subclasses and Heirtypes

Naive definitions of subclasses can easily come to our minds, such as "subalgebra" and "subset". However, it immediately turns out that these definitions are too restrictive to explain subclasses in OOPLs. For example, class *student(name, age, class)* is a subclass of *person(name, age)*. This relation cannot be defined by either subalgebra or subset. In this subsection, we address the issue of how to model subclasses in the framework of axiomatized structured signatures. We start by a few definitions.

Definition 6.18 Let Σ and Σ' be two axiomatized structured signatures, and δ be an axiomatized-signature morphism from Σ to Σ', where $\delta = (\delta_1, \delta_2, \delta_3, \{\delta_\varsigma\}_{\varsigma \in \mathcal{E}(\Sigma)})$. The *image* of Σ under δ, denoted $\delta(\Sigma)$, is defined as follows:

- $\mathcal{S}(\delta(\Sigma)) = \delta_1(\mathcal{S}(\Sigma)) = \{\delta_1(\varsigma) | \varsigma \in \mathcal{S}(\Sigma)\}$, where $\delta_1(\varsigma)$ is the image of ς under δ_ς if $\varsigma \in \mathcal{E}(\Sigma)$;
- $\mathcal{F}(\delta(\Sigma)) = \delta_2(\mathcal{F}(\Sigma)) = \{\delta_2(f) | f \in \mathcal{F}(\Sigma)\}$;
- $\mathcal{M}(\delta(\Sigma)) = \delta_3(\mathcal{M}(\Sigma)) = \{\delta_3(e) | e \in \mathcal{M}(\Sigma)\}$.

Note that δ is supposed to be injective for practicality. Hence, $\delta(\Sigma)$ is axiomatized-signature isomorphic to Σ. Σ' is called a *horizontal extension* of Σ, denoted $\Sigma \xrightarrow{h} \Sigma'$, if $\Sigma = \delta(\Sigma)$; otherwise, Σ' is called a *vertical extension* of Σ, denoted $\Sigma \xrightarrow{v} \Sigma'$.

Definition 6.19 Let Σ and Σ' be two axiomatized structured signatures, and δ be an axiomatized-signature morphism from Σ to Σ'. Σ'-algebra A' is said to be an *h-heir* algebra of Σ-algebra A, denoted $A \xrightarrow{h} A'$, if the following conditions hold:

(1) $A = A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}$,

(2) Σ' is a horizonal extension of Σ.

It is obvious that A' satisfying (1) and (2) above can be obtained directly from A by adding more features.

Definition 6.20 Let Σ and Σ' be two axiomatized structured signatures, and δ be an axiomatized-signature morphism from Σ to Σ'. Σ'-algebra A' is said to be a *v-heir* algebra of Σ-algebra A, denoted $A \xrightarrow{v} A'$, if the following conditions hold:

(1) $A = A'|_{\Gamma(\delta), c_{\Gamma(\Sigma)}}$,

(2) Σ' is a vertical extension of Σ.

Obviously, A' can be obtained from A by extending some algebra domains and renaming some operations.

According to Theorem 6.4, any Σ'-algebra is an h-heir algebra of some Σ-algebra if Σ' is a horizonal extension of Σ, and, any Σ'-algebra is a v-heir algebra of some Σ-algebra if Σ' is a vertical extension of Σ. We call Σ' a *subsignature* of Σ if $\Sigma \xrightarrow{h} \Sigma'$ or $\Sigma \xrightarrow{v} \Sigma'$.

Σ-classes are special kinds of Σ-algebras that can be finitely generated. Hence, Σ'-class A is called a *subclass* of Σ-class B if $B \xrightarrow{h} A$ or $B \xrightarrow{v} A$. Σ'-type T' is called an *heirtype* of Σ-type T if any Σ'-class in T' is a subclass of some Σ-class in T.

6.3.5 Kinds

Now we define kinds in our algebraic framework. Since "$\xrightarrow{h}$" is a special case of "$\xrightarrow{v}$", we simply use $\Sigma \xrightarrow{c} \Sigma'$ to denote that $\Sigma \xrightarrow{h} \Sigma'$ or $\Sigma \xrightarrow{v} \Sigma'$. Obviously, "$\xrightarrow{c}$" is a reflexive and transitive relation. For any axiomatized structured signature Σ, we define a set of class signatures as follows:

$$\Sigma^* = \{\Sigma' | \Sigma' \in ClassSign,\ \Sigma \xrightarrow{c}{}^* \Sigma'\}$$

where "$\xrightarrow{c}{}^*$" is the reflexive and transitive closure of "$\xrightarrow{c}$". That is, Σ^* includes all class signatures that can be derived from Σ under the condition of axiomatized-signature morphism. A Σ'-class is said to be a Σ^*-class if $\Sigma' \in \Sigma^*$. Thus, Σ^* defines the following set of Σ^*-classes:

$$Class(\Sigma^*) = \bigcup_{\Sigma' \in \Sigma^*} Class(\Sigma')$$

Obviously, any Σ^*-class in $Class(\Sigma^*)$ has the properties that are specified in Σ. Hence, Σ^* is called a *kind* with respect to Σ.

Now we have the following correspondence between our four-level type hierarchy and the object-oriented algebraic theory. For any axiomatized structured signature Σ, a Σ-class corresponds to a class, and a Σ-type corresponds to a type, and Σ^* corresponds to a kind.

6.4 Remark

We have presented an object-oriented algebraic framework to model objects, classes, types, and kinds in our four-level type hierarchy. In an axiomatized structured signature, an interior sort is explicitly distinguished from exterior sorts, and an exterior sort can be another axiomatized structured signature. This extension makes it possible to explain our four-level type hierarchy in the algebraic framework.

We have shown that the semantics of a class signature Σ is a set of Σ-types, and the semantics of a Σ-type is a set of isomorphic Σ-classes. In fact, there are three main ways in which we can define the semantics of a class signature, i.e., *loose* approach, *initial approach*, and *terminal approach.*

In the loose semantics, all types in $Type(\Sigma)$ are taken as the interpretation of class signature Σ. Obviously, this approach is appropriate for our purpose. However, to be able to specify a unique data type in practice, it is necessary to choose isomorphic Σ-classes.

In the initial semantics, only the initial Σ-type is chosen as the interpretation of Σ. In the terminal semantics, the interpretation of Σ is the terminal Σ-type. In both cases, a class signature is regarded as a type since it specifies a set of isomorphic classes.

A class signature is not only a syntatic specification but also a behavioural specification for a group of classes. Based on the object-oriented algebraic framework presented in this chapter, we have defined an object-oriented specification language (OOSL) in Appendix A. This language can be embedded in programming languages to guide the development of classes. For example, it has been used in Kind-C++ in the form of comments.

Chapter 7

Conclusions

The development of algorithmic abstraction at the programming language level has been the goal of this thesis. The approach pursued relies strictly on (1) a four-level type hierarchy that supports the definition of algorithm domains and (2) expressive forms of polymorphism that allow us to write sequences of steps that are polymorphic to all types in an algorithm domain. We have shown that kinds can be used to define the domains of algorithms, and that a kind-bounded function is essentially an algorithm at the programming language level.

Due to the wide range of subjects, many interested issues could not been studied in full detail. We end this thesis with a list of contributions, a comparison of our work with other related works, and a couple of issues for future work. These are followed by final remarks.

7.1 Contributions

The contributions of this thesis can be summarized as follows:

1. We introduced the concept of algorithmic abstraction in programming languages. Algorithmic abstraction is different from data abstraction although the former is based on the latter.

2. We clarified the distinction between classes and types. A type is a set of classes that have exactly the same external behaviour, and a class is a specific implementation of a type. But classes and types are used as interchangeable concepts

in many object-oriented programming languages.

3. We introduced the concept of kinds for specifying the algorithm domains in programming languages. A kind is a set of types that satisfy a common set of properties, and an algorithm demands certain properties on each type of its domain. Hence, kinds can be used to specify the algorithm domains properly.

4. We introduced kind-bounded polymorphism for specifying the step sequences of algorithms. Universal-bounded polymorphism and subtype-bounded polymorphism are only two special cases of kind-bounded polymorphism because both *TOP* and *Power*(τ) are kinds. Hence, kind-bounded polymorphism is the most general form of parametric polymorphism.

5. We introduced structured signatures and structured algebras for the semantics of the four-level type hierarchy. An exterior sort in a structured signature can be another structured signature, and a carrier domain in a structured algebra can be another structured algebra. This is a significant departure from traditional signatures.

6. We implemented KC for supporting algorithmic abstraction. KC is essentially an extension of C++ with kinds and kind-bounded polymorphism.

7.2 Comparison

The four-level type hierarchy we have presented is adequate for representing algorithm domains. The strength of the type hierarchy is reflected in KC (Kind-C++), which is an extension of C++ with kinds. Kinds in KC are similar to type signatures in Russell [22], type classes in Haskell [24], and signatures in G++ [5], etc. However, we explain in the following that our notion of kinds in KC is different from each of the above.

In essence, Russell supports a two-level type hierarchy, i.e., values and signatures, rather than a four-level type hierarchy. In Russell, one cannot define a set of objects that have exactly the same internal representation (data structures and methods).

In Haskell (a pure functional language), type classes can be defined in terms of unrestricted type variables. Hence, Haskell essentially realizes universal quantification. Thus, type classes in Haskell are less suitable for defining restrictions on complicated types than kinds.

Both signatures in G++ and types in POOL-I ([2]) suffer from the same problem as abstract classes in C++. They cannot define the following type restrictions: the two parameters of a polymorphic function can be both of type T_1 or both of type T_2 but should not be of T_1 and T_2, respectively. In fact, signatures in G++ play the role of types [5]. Hence, its informal signature-checking has the problem implied in bounded quantification.

Modula-3 supports generic interfaces and generic modules which allow us to define families of interfaces and modules, respectively [34]. However, formal imports (i.e., formal parameters) in generic interfaces (modules) are unrestricted. Hence, the correct instantiation of a generic interface (module) requires that users read through the entire code of the generic interface (module). Similarly, Eiffel supports generic classes by using unrestricted type variables [41, 43]. Hence, Eiffel suffers from the same problems as Modula-3.

The *where* clause of parameterized procedures in CLU ([39]) and the *with* statement of generic procedures in Ada ([54]) are all useful and convenient constructs for implementing constrained polymorphism. However, unlike kinds, the *where* clause and the *with* statement are not independent entities that are one level higher than types in their type structures. For example, the *where* clause in CLU is treated as an abbreviation for a lengthy textual representation. In contrast, kinds are treated uniformly as other entities in our type hierarchy. A type variable restricted to a kind is the same as a value variable restricted to a class (or a type) except at a higher level. Especially for C++, the introduction of kinds is consistent with its type structure. Adding any other ad hoc constructs, e.g., "with" or "where", to C++ would make the language awkward.

We should also mention OBJ2 [26]. In OBJ2, the only top level OBJ2 entities are modules (which are either objects or theories) and views (which relate theories to modules). Objects contain executable code, while theories contain non-executable assertions. The notions of a kind and a theory look similar. However, theories are at the same level as 'objects' (classes) in OBJ2, but kinds are at a higher level than classes in KC. Clearly, the relationship between kinds and classes is more transparent.

The notion of kinds is also used in Quest [16]. By introducing kinds, Cardelli intended to unify the two forms of polymorphism: parametric polymorphism and inclusion polymorphism. Unfortunately, the notion of kinds was not fully developed in [16]. Only three types of kinds can be defined: the universal kind, which is the set of all types, the power kind, which is the set of all subtypes of a type, and the function kind. In contrast, a *kind* in our type hierarchy can be defined by an arbitrary set of properties.

7.3 Future Work

In Chapter 6, an object-oriented algebraic theory is presented to model objects, classes, types, and kinds. For simplicity, we have concentrated on structured algebras that satisfy some set of equations. The simplification is based on the fact th*t a great number of data types can be specified within the equational framework [30]. It seems that this restriction could be dropped without affecting the results established in the chapter. However, a complete proof needs to be done. In particular, a special technique is needed to handle the inequality since two different configu. ations of a Σ'-algebra A' may be coerced into the same configuration in the Σ-algebra $A'|_{\Gamma(\delta),c_{\Gamma(\Sigma)}}$, where δ is an axiomatized morphism from Σ to Σ'.

Inheritance has been modelled in the object-oriented algebraic framework. However, it is still not sufficient to model subtype. Hence, more research is needed. One possibility is to introduce an order on the exterior sorts of a structured signature just like order-sorted signatures [29, 7]. Then, subtype could be modelled in the way similar to t' e one used by Bruce and Wegner [11].

7.4 Final Remarks

Algorithmic abstraction is a totally different concept from data abstraction although the former is developed based on the latter. Implementing algorithmic abstraction in object-oriented languages has as least the following advantages:

(1) It will fundamentally change the way of programming, which will become more like program building than program writing. Algorithms designed by algorithm designers are stored in algorithm libraries, and programmers use them directly without any recoding. In other words, it will increase the reusability of subprograms.

(2) It will encourage the use of efficient algorithms and reduce the latency period between the design of an algorithm and its prac'ical use.

(3) It will affect algorithm design, too. Algorithm designers have to be more precise about their algorithm domains and polymorphic operators used in their algorithms. For example, it should be specified clearly for each parameter what is the minimum set of properties that are required by an algorithm. Those problems are usually not explicitly stated ir ~urrent algorithm descriptions.

(4) It will also increase the reliability of softw..re, and program verification will become more realistic and manageable.

Appendix A

OOSL – A Class Specification Language

In this appendix, we provide a minimal object-oriented specification language called OOSL for class specifications. OOSL is based on the algebraic framework we have presented in Chapter 6. Hence, the semantics of OOSL is trivial. OOSL offers basic operations for hierarchical constructions of class specifications. We first present the syntax of OOSL. Then we give an example to demonstrate how this language works.

A.1 The Syntax of OOSL

$$
\begin{array}{lcl}
<\mathit{classspec\_envi}> & ::= & <\mathit{classspec}> \mid <\mathit{class\_envi}>; <\mathit{classspec}> \\
<\mathit{classspec}> & ::= & \textbf{ClassSpec} <\mathit{classspec\_id}> \textbf{is} \\
 & & \qquad\qquad <\mathit{classspec\_exp}> \\
 & & \textbf{EndSpec} \\
<\mathit{classspec\_exp}> & ::= & \{<\mathit{use\_clause}>\}_0^1 \\
 & & \{<\mathit{extend\_clause}>\}_0^1 \\
 & & \{<\mathit{opn\_clause}>\}_0^1 \\
 & & \{<\mathit{axiom\_clause}>\}_0^1 \\
<\mathit{use\_clause}> & ::= & \textbf{use}\ \{<\mathit{classspec\_id}>\}_1^n \\
<\mathit{extend\_clause}> & ::= & \textbf{extend}\ \{<\mathit{extended\_exp}>\}_1^n\ \{\textbf{with}\}_0^1 \\
<\mathit{opn\_clause}> & ::= & \textbf{operations}\ \{<\mathit{operation}>\}_1^n \\
<\mathit{axiom\_clause}> & ::= & \textbf{axioms}\ \{<\mathit{axiom}>\}_1^n \\
<\mathit{extended\_exp}> & ::= & <\mathit{classspec}>\ \{<\mathit{renameing\_exp}>\}_0^n \\
<\mathit{renaming\_exp}> & ::= & \textbf{rename by}\ (\{<\mathit{renaming}>\}_1^n)
\end{array}
$$

```
< renaming >       ::= < operation_id >→< operation_id >
< operation >      ::= < operation_id >:< operation_type >
< operation_type > ::= < classspec_id > | (< classspec_id >) |
                       < input_type >→< output_type >
< input_type >     ::= < classspec_id > |(< classspec_id >) |
                       < input_type > × < classspec_id > |
                       < input_type > ×(< classspec_id >)
< output_type >    ::= < classspec_id > |(< classspec_id >)
< axiom >          ::= define as before
< classspec_id >   ::= < identifier >
< operation_id >   ::= < identifier >
< identifier >     ::= a string starting with a letter
```

Here, $\{e\}_m^n$ means that the number of occurrences of e should be between m and n. The **use** clause introduces exterior signatures which are used as base signatures of the signature being defined. The **extend** clause defines the signatures from which the signature being defined is derived. The **rename** clause introduces the renaming operations. If a class specification C does not contain any **use** or **extend** clause, then C is said to be *basic*.

Note that OOSL supports class specification inheritance using the **extend** clause. Since an **extend** clause can introduce more than one signature, multiple inheritance is also possible in OOSL.

A.2 An Example

Now we use sets of natural numbers as an example to demonstrate the use of OOSL. For simplicity, we assume that a set of natural numbers has only methods *newset*, *empty*, and *insert*. Thus, we can specify the set classes with the following class signature:

ClassSpec *nset* **is**

use *bool*, *nat*

operations

$newset : \ \rightarrow (nset)$

$empty : (nset) \rightarrow bool$

$insert : (nset) \times nat \rightarrow (nset)$

axioms

$\forall n \in nat, \forall s \in (nset)$

$empty(insert(s, n)) = false$

$empty(newset) = true$

EndSpec.

There is no order on the elements of a set. Hence, it is not important where an element is inserted. *nat* and *bool* are two base signatures of *nset*, which are defined as follows:

ClassSpec *nat* =

operations

$zero : \ \rightarrow (nat)$

$succ : (nat) \rightarrow (nat)$

EndSpec

ClassSpec *bool* =

operations

$true \ : \ \rightarrow (bool)$

$false : \ \rightarrow (bool)$

EndSpec.

A list is a special set in which duplicate elements are allowed. In addition, there is an order on the elements of a list. Thus we can define a class specification for lists of natural numbers from *nset* as follows:

ClassSpec *nlist* = **extend** *nset* **rename by**

($(nset) \rightarrow (nlist)$,

$newset \rightarrow newlist$

)

with

operations

$first : (nlist) \rightarrow nat$

$rest : (nlist) \rightarrow (nlist)$

axioms

$\forall n \in nat, \forall s \in (nlist)$

$first(insert(s, n)) = n$

$rest(insert(s, n)) = s$

EndSpec.

A stack is a special list where insertion and deletion happen to the same end. We can specify stacks of natural numbers from *nlist* as follows:

ClassSpec *nstack* = **extend** *nlist* **rename by**

($(nlist) \rightarrow (nstack)$

$insert \rightarrow push$

$first \rightarrow top$

$rest \rightarrow pop$

)

EndSpec.

REFERENCES

[1] A. V. Aho and J. D. Ullman. *Foundations of Computer Science.* Computer Science Press, New York, 1992.

[2] P. America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In *Lecture Notes in Computer Science* **489, pages 60–89.** Springer-Verlag, 1991.

[3] J. G. P. Barnes. *Programming in Ada.* Addison_Wesley Publishing Company, third edition, 1989.

[4] F. L. Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[5] G. Baumgartner and V. F. Russo. Signatures: A C++ Extension for Type Abstraction and Subtype Polymorphism. Technical Report CSD-TR-93-059, Department of Computer Science, Purdue University, 1993.

[6] J. Bergin. *Data Abstraction: The Object-Oriented Approach Using* C++. McGraw-Hill, Inc., 1994.

[7] R. Breu. Algebraic Specification Techniques in Object Oriented Programming Environments. In *Lecture Notes in Computer Science* **562, pages 1–228.** Springer-Verlag, 1991.

[8] K. B. Bruce and G. Longo. A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation,* **87**:196–240, 1990.

[9] K. B. Bruce and A. R. Meyer. The Semantics of Second Order Polymorphic Lambda Calculus. In *Lecture Notes in Computer Science* **173**, pages 131–143. Springer-Verlag, 1984.

[10] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The Semantics of Second-Order Lambda Calculus. *Information and Computation*, **85**:76–134, 1990.

[11] K. B. Bruce and P. Wegner. An Algebraic Model of Subtype and Inheritance. In *Advances in Database Programming Languages*, pages 75–96. Press Frontier Series, 1990.

[12] R. M. Burstall and J. A. Goguen. Putting Theories Together to Make Specifications. In *Proc. 5th Internat. Joint Conf. on Artificial Intelligence*, pages 1045–1058. 1977.

[13] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded Polymorphism for Object-Oriented Programming. In *Proc. 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[14] L. Cardelli. A Semantics of Multiple Inheritance. In *Lecture Notes in Computer Science* **173**, pages 51–68. Springer-Verlag, 1984.

[15] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, **76**:138–164, 1988.

[16] L. Cardelli. Typeful Programming. In *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, 1991.

[17] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, **17**(4):471–522, 1985.

[18] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, Princeton, N.J., 1941.

[19] A. Church and J. B. Rosser. Some Properties of Conversion. *Trans. Amer. Math. Soc.*, **39**:220–232, 1936.

[20] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance Is Not Subtyping. In *Proc. 17th ACM Symp. on Principle of Programming Languages*, pages 125–135, 1990.

[21] H. B. Curry. Grundlagen der kombinatorischen logik. *American J. Math.*, **52**:509–636, 1930.

[22] J. Donahue and A. Demers. Data Types Are Values. *ACM Transactions on Programming Languages and Systems*, **7**(3):426–445, 1985.

[23] M. A. Ellis and B. Stroustrup. *The Annotated* C++ *Reference Manual*. Addison_Wesley, Reading, 1990.

[24] P. Hudak *et al.* A Non-Strict, Purely Functional Language. *ACM SIGPLAN Notices*, **27**(5):R1–R164, 1992.

[25] J. McCarthy *et al.* *The Lisp 1.5 Programmer's Manual.* MIT Press, Cambridge (Mass.), 1962.

[26] K. Futasugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. ACM Symp. on Principle of Programming Languages*, pages 52–66, 1985.

[27] J.-Y. Girard. Une extension de l'interpretation de Gödel *à* l'analyse, et son application *à* l'élimination des coupures dans l'analyse et la théorie des types. In *Proc. Second Scandinavian Logic Symposium*, 1971.

[28] J.-Y. Girard. Interpretation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. *Thése D'Etat, Université Paris VIII*, 1972.

[29] J. A. Goguen and J. Meseguer. Order-Sorted Algebra I : Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. *Theoretical Computer Science*, **105**(2):217–273, 1992.

[30] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology* **4**. Prentice-Hall, 1978.

[31] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and Its Implementation.* Addison_Wesley, Reading, Mass., 1983.

[32] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* Ph.D Thesis, Department of Computer Science, University of Toronto, Tech. Report CSRG-59, 1975.

[33] J. V. Guttag. Abstract Data Types and the Development of Data Structures. *CACM*, **20**:396–404, 1977.

[34] S. P. Harbison. *Modula-3.* Prentice-Hall, Inc., 1992.

[35] J. R. Hindley, B. Lercher, and J. P. Seldin. *Introduction to Combinatory Logic.* Cambridge University Press, 1972.

[36] J. R. Hindley and J. P. Seldin. *Introduction to Combinatory Logic and λ-Calculus.* Cambridge University Press, 1986.

[37] E. Horowitz and S. Sahni. *Fundamentals of Data Structures in Pascal.* Computer Science Press, 1987.

[38] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, **24**(2):131–183, 1992.

[39] B. Liskov. CLU Reference Manual. In *Lecture Notes in Computer Science* **114**, pages 1–190. Springer-Verlag, 1981.

[40] B. Liskov and S. N. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, **9**(4):50–60, 1974.

[41] G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Object-Oriented Languages.* Academic Press, 1991.

[42] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Comm. ACM*, **3**(4):158–165, 1960.

[43] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall International Ltd., 1988.

[44] R. Milner. A Proposal for Standard ML. In *Symposium on Lisp and Functional Programming*, pages 184–197, 1984.

[45] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT PRESS, 1990.

[46] J. Mitchell. Type Inference and Type Containment. In *Lecture Notes in Computer Science* **173**, pages 257–277. Springer-Verlag, 1984.

[47] R. Morrison, A. Dearle, R. C. [illegible]onnor, and A. L. Brown. An Ad Hoc Approach to the Implementation of Polymorphism. *ACM Transaction on Programming Languages and Systems*, **13**(3):342–371, 1991.

[48] G. Revesz. *Lambda-Calculus, Combinators, and Functional Programming.* Cambridge University Press, 1988.

[49] J. C. Reynolds. *The Craft of Programming.* Prentice-Hall International, Inc., 1981.

[50] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt. An Introduction to Trellis/Owl. In *Proc. ACM Cof. on Object-Oriented Programming: Systems, Languages and Applications*, pages 6–16, 1986.

[51] P. G. N. Scheyen. *New Constructs For Implementing Generic Functions and Parameterized Classes in* C++. Master thesis, Department of Computer Science, The University of Western Ontario, 1994.

[52] C. Strachey. Fundamental Concepts in Programming Languages. Lecture Notes from International Summer School in Computer Programming, Copenhagen, 1967.

[53] B. Stroustrup. C++ *Programming Language*. Addison_Wesley Publishing Company, second edition, 1991.

[54] Ada 9X Mapping/Revision Team. Ada 9X Reference Manual. Intermetrics, Inc., 1994.

[55] Paul Wang. *C++ with Object-oriented Programming*. PWS Publishing Co. Boston, MA., 1994.

[56] A. Wikstrom. *Functional Programming Using Standard ML*. Prentice Hall, first edition, 1987.

[57] M. Wirsing. Algebraic Specification : Semantics, Parameterization and Refinement. In *Formal Description of Programming Concepts*, pages 259–318. Springer-Verlag, 1991.

[58] M. Wirsing, P. Pepper, and H. Partsch. On Hierarchies of Abstract Data Types. *Acta Inform.*, **20**:1–33, 1983.

[59] S. Yu and Q. Zhuang. Software Reuse via Algorithm Abstraction. In *Proc. 17th International Conference on Object-Oriented Languages and Systems*, pages 277–292, 1995.

[60] S. Yu and Q. Zhuang. Algorithmic Abstraction in Object-Oriented Languages. *Object-Oriented Systems*, accepted.

[61] K. Zhang and D. Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. on Computing*, **18**(6):1245–1262, 1989.

[62] Q. Zhuang. Behaviour-Bounded Inclusion Polymorphism for Object-Oriented Languages. In *Proc. 5th International Conference for Young Computer Scientists*, pages 375–380, 1995.