Diplomarbeit

Subtyping and Inheritance in Object-Oriented Programming

ausgeführt am Institut für Computersprachen

der Technischen Universität Wien

unter der Anleitung von

O.Univ.Prof. Dipl.-Ing. Dr.techn. Manfred Brockhaus

und Univ.Ass. Dipl.-Ing. Dr.techn. Franz Puntigam

als verantwortlich mitwirkendem Universitätsassistenten

durch

Stoyan Kurtev

Mühlwasserpromenade 23/9, A-1220 Wien

7. Februar, 2000

Abstract

Subtyping and inheritance belong to the major concepts in contemporary object-oriented programming languages, their presence having decisively influenced the success of the object-oriented paradigm. Inheritance constitutes a technique for reusing code from previous endeavours, while subtyping establishes semantic relationships between object behaviors. Although these, in principle, independent and autonomous notions are rooted in different domains of programming, their distinction is poorly understood at large. The necessity is recognized to investigate the notions of types, implementations, subtyping, and inheritance in theory and practice. After presenting these concepts in a language-independent fashion, we contrast subtyping with inheritance, and elaborate the integration of these mechanisms in two different object-oriented programming languages. We conclude that the separation is essential both in strongly-typed and dynamically-typed programming languages, facilitating the design in terms of object behaviors to be distinguished from implementation.

Kurzfassung

Subtypen und Vererbung gehören zu den wichtigsten Konzepten in der objekt-orientierten Programmierung und haben wesentlichen Anteil an der Popularität des objekt-orientierten Paradigmas in der modernen Softwareentwicklung. Während Vererbung ein Mechanismus zur Wiederverwendung von vorhandenen Implementierungen ist, stellen Subtypen semantische Beziehungen zwischen den externen Verhaltensweisen von Objekten dar. Obwohl diese eigenständigen Konzepte in gänzlich verschiedenen Teilgebieten der Programmierung verwurzelt sind, werden sie, im allgemeinen, nur mangelhaft differenziert. Die Notwendigkeit entsteht, eine Analyse von Typen, Implementierungen, Subtypen und Vererbung aus theoretischer und praktischer Sicht durchzuführen. Nach einer sprachunabhängigen Präsentation dieser Konzepte werden Subtypen und Vererbung einander gegenübergestellt, sowie deren Integration in zwei unterschiedlichen objekt-orientierten Programmiersprachen untersucht. Dabei wird die Wichtigkeit einer klaren Trennung sowohl in stark als auch in dynamisch typisierten Sprachen festgestellt, um eine Unterscheidung zwischen verhaltens-orientiertem Entwurf und code-orientierter Implementierung zu ermöglichen.

Contents

Chapter 1

Introduction

Two of the most intriguing concepts in object-oriented design and implementation are the notions of subtyping and inheritance. Inheritance constitutes a powerful mechanism for structuring large programs, aiming at the reduction of the overall implementational effort and improvement of a program's reusability, maintainability, and flexibility. On the other hand, subtyping establishes a semantic relationship between the behavior of objects in a program. In general, while the unequivocal separation between these concepts is adequately understood by object-oriented researchers, the clear distinction is poorly understood by object-oriented practitioners at large.

This is, in part, due to the fact that the importance of inheritance is frequently overemphasized in object-oriented programming. Inheritance is commonly regarded as the feature most central to object-orientation, supposedly being the only notion critically associated with object-oriented programming and distinguishing the object-oriented paradigm from other programming paradigms. In practice, inheritance has always received considerably more attention than subtyping. Following this ubiquity, the notion of inheritance has assimilated the notion of subtyping, making inheritance a rather controversial mechanism.

Consequently, although the separation between subtyping and inheritance is widely acknowledged in research, a majority of the modern object-oriented programming languages like C++ [Str97] and Eiffel [Mey97] have unified the formerly distinct concepts into a single mechanism. This identification has immediate and far-reaching implications in terms of the expressiveness and flexibility of object-oriented programming.

1.1 Goal and Methodology

The aim of this work is to give a clearer and more uniform view of fundamental concepts in object-oriented programming and practice, attempting to alleviate various well-known and notorious controversies. The study presents a comprehensive exposition of the preferences and weaknesses of identifying subtyping with inheritance. A major concern must be to introduce a precise terminology, precluding any ambiguities in the discussion. In particular, the necessity is recognized to investigate the notions of types, implementations, subtyping, and inheritance in theoretical terms and in practice, founding the discussion on a well-established type-theoretic basis and simultaneously making these important issues accessible to practitioners.

We aim to elucidate the notion of subtyping as a client's view of a program, based on the behavior of the objects involved. This view must be clearly distinguished from that induced by inheritance, an implementational technique concerned with the sharing of code between object implementations. To obtain an extensive understanding of subtyping and inheritance, these concepts should be compared both on theoretical grounds and in their respective outgrowths in existing object-oriented programming languages. In other terms, these mechanisms must be associated with precise definitions and concepts, facilitating a detailed and exhaustive comparative analysis.

The approach taken in this study is to initially present a language-independent and simple theory of types, implementations, subtyping, and inheritance. The general proceeding to achieve language-independence is to define each concept in abstract terms, without reference to specific aspects of programming languages. This abstract definition is subsequently rendered to more concrete mechanisms in a fictional object-oriented language. Thus, types and subtyping are defined in terms of abstract predicates, which are later mapped to predicates describing concrete objects. Similarly, inheritance is defined as a general mechanism facilitating the sharing of abstract program entities, and later is specialized to implementations and types, respectively.

The second step is to identify types with implementations, and subtyping with inheritance, and to determine the ramifications of this unification on the expressiveness of object-oriented programming. These restrictions are contrasted with the expressiveness of separate subtyping and inheritance hierarchies. In particular, we search for type-safe approaches to minimizing these constraints.

Finally, we elaborate the importance and integration of these mechanisms in contemporary object-oriented programming languages. We will investigate to what extent the presented theory can be applied to (or be simulated in) two profoundly different object-oriented languages, namely C++ [Str97] as a representative strongly-typed language, and *Susi* [Kur99] as an exploratory dynamically-typed language[1]. A pervasive comparison of the theoretical issues with C++ and *Susi* will reveal new insights into implementing maintainable and flexible software in both strongly-typed and dynamically-typed object-oriented programming languages.

1.2 Outline

The structure of this study can be outlined as follows. The next chapter provides an abstract definition of types and subtyping, and applies these definitions to functions and objects, respectively. Further, the issue of types in dynamically-typed languages is treated, accompanied by a discussion of structural and named type equivalence and subtyping. Chapter 3 introduces the notion of implementations and establishes a relationship between implementations and types. Chapter 4 defines an abstract inheritance mechanism, based on the reuse of abstract and recursive program entities. This abstract mechanism is rendered to the reuse of implementations and type specifications. Chapter 5 contrasts the restrictions imposed by unifying subtyping and inheritance to the enhanced expressiveness of separate hierarchies. Chapter 6 explores the ramifications of type-safe contravariant specialization on object-oriented programming, and discusses a statically-sound alternative based on overloaded functions and

[1] The language *Susi* has been designed and implemented by the author, attempting to produce a full-fledged object-oriented language founded on a minimum of concepts. A description of *Susi* is included in Appendix A.

multiple dispatching. Chapter 7 exhaustively compares the discussed theoretical issues with C++ and *Susi*, summarizing with a tabular overview of the supported features. Finally, conclusions are presented.

We would like to emphasize that the language constructs used to illustrate our examples are in no terms proposals for some specific object-oriented language. Rather, these constructs are merely used as a vehicle for conveying certain concepts. In this sense, the concrete syntax in the examples is a negligible issue. Further, the examples are intentionally kept short and simple, posing no attempt to present a complete or optimal solution to a given problem. The examples are contrived to illustrate a specific mechanism.

1.3 Related Work

The presentation in [LP91] contrasts the concepts of subtyping and inheritance in a language-independent and comprehensible fashion. However, the discussion has no theoretic background and the practicability is largely confined to Smalltalk-80. Similarly, the approaches in [AL90, KMN+90, Por92, KLMM94] address the clear distinction between subtyping and inheritance, but illustrate these concepts only in one particular language, not attempting to provide a more general exposition. Specific languages are informally compared in [Sei87, BPS89, AGS91, SJ94, Whe97, Bro98], however only from one very specific viewpoint.

More formal approaches to subtyping and inheritance such as [CW85, Car88, CP89, CHC90] are necessarily couched in theoretical terms, investigating the fundamental features of object-oriented programming by expressing them in terms of elegantly typed calculi, consequently staying inaccessible to practitioners. [CCHO89] and [CHC90] are more concerned with class recursion than with contrasting subtyping and inheritance, giving only brief comparisons to existing programming languages. In particular, [CCHO89] presents a clear distinction between interfaces and implementations, but falls short in distinguishing the concepts of subtyping and inheritance in practice.

An exhaustive study of a majority of the nuances of inheritance can be found in [Tai96]. However, the differentiation with subtyping is essentially confined to the reproduction of a few papers [Ame87, Ame90, WZ88, LP91]. On the contrary, [Sou97] starts off with a language-independent view, presenting types and implementations as autonomous notions, and maps these general concepts to concrete programming languages like Java, C++, and Smalltalk-80. Unfortunately, the brevity of this contribution fails to provide the necessary substance and depth for a more comprehensive debate.

Chapter 2

Types and Subtyping

Since subtyping and inheritance essentially describe relationships between types and implementations, respectively, a prerequisite for a clear distinction between subtyping and inheritance is a founded understanding of types and implementations. The following sections informally introduce these concepts and demonstrate how they are related to one another in the domain of object-oriented programming.

2.1 Types

Although the term "type" has undergone many changes during the past few decades, evolving from a means of determining efficient memory layout for compilers to a powerful classification tool for software engineering, a common understanding of types[1] can be obtained by viewing them as *predicates* on values [PS92], or, in the context of object-oriented programming, on objects. There exists a wide spectrum of suggestions for predicates, corresponding to different levels of specificity for describing types. These proposals range from requiring the existence of a particular set of named methods to imposing restrictions on the behavior of the required methods [WZ88, CCHO89, Cook90, Cook92].

Generally, a type defines a set of values—the set of values satisfying the associated predicate[2]. In the case of object-oriented programming, an object x belongs to a type σ if x satisfies the predicate corresponding to σ. As we shall see later, this definition naturally coincides with polymorphic type systems, since the sets of objects associated with different types may intersect, and therefore, in general, an object can have many, possibly incomparable types.

When considering the type of a variable or expression, the programmer is typically not interested in how the object is represented internally, but rather in how the object can be *used* [Ame90]. Thus, types should provide the information necessary to properly use an object. In particular, distinct objects with common behavior should be logically interchangeable [Budd91]. For example, the user of a stack object should usually not be concerned with whether the stack behavior is implemented in terms of an array or a linked list.

[1]We regard types as static entities which describe immutable object properties. See [Cha93, Rum93] for an alternative view.

[2]If the predicate is not applicable to a specific value, then we assume it to evaluate to *false*.

In object-oriented systems, the only possible interaction with an object, apart from assignment to variables or formal parameters, is sending it a message. Therefore, the only relevant information for typing purposes is how an object reacts to messages [CCHO89]. A type should provide a behavioral specification of an object's operations at some level of abstraction [Budd91, LP91]. In particular, the internal details of an object, such as the names of its instance variables, are not important. We choose to settle with the following definition [Ame87, Ame90, LP91, AC93, Sou97]:

Type: A type τ describes a collection of values that share some *externally observable behavior*, in terms of observable by sending messages. In principle, a type should not provide any information about the implementations of its elements[3]. For each type τ there exists an associated predicate ψ_τ, specifying the behavior of τ's elements, such that $\psi_\tau(\mathrm{x})$ for all values $x \in \tau$. The behavior of τ's elements may, in general, be specified in any arbitrary kind of formal or informal logic.

Example 2.1: Informal description of the type *LockableIntegerBuffer*.
An object of this type accepts the messages *put*, *get*, *lock*, and *unlock*. The message *put* takes an integer argument. The message *get* returns an integer previously inserted using *put*. The type poses no restrictions on the retrieval order of the stored integers. The message *get* is only accepted if the number of *put* messages already accepted exceeds the number of *get* messages accepted. The only message acceptable after *lock* is *unlock*.

While such descriptions may be useful in early stages of design, they are clearly intended solely for human readers and not well-suited for automated processing. Ideally, a type should provide a complete and formal specification of all interesting aspects of an object's externally observable behavior [Sny86, Ame87]. In deciding whether a specific object belongs to a given type, we are typically interested in information such as

- the messages which the object can respond to (derivable from the object's methods),
- the order in which these messages may be sent to the object (for example, synchronization constraints in concurrent object-oriented programming describe the conditions for sending certain messages [Frø92, CRR98]),
- and possible restrictions on the values of the arguments as well as possible result values (in relation to the argument values).

Given such a behavioral specification, connections between modules could be validated by theorem proving [CCHO89]. Unfortunately, formal specification technology and mechanical theorem proving in particular are not evolved enough to include complete formal specifications into practical programming languages [CCHO89, DT92]. However, formal specifications can substantially contribute to the understanding of mechanisms like subtyping [Wing87, LW90, LW94].

[3] It should be emphasized that since types describe externally observable behavior, they are independent of concrete implementations. Object implementations are discussed in Chapter 3. Objects with different implementations may still belong to one and the same type. However, note that our definition of types in terms of predicates is actually general enough to encompass types as found in C++, which *are* coupled to specific implementations (see Section 5.1).

In general, a type will not be able to capture all nuances of an object's behavior [LP91]. Types are typically only *partial* specifications [Car91]. Notably, one of the most important aspects of behavior is whether an object can accept a given message at all [Ame90]. Since messages are ultimately associated with methods, this aspect can easily be expressed by describing the object's available methods in terms of *signatures*:

Signature: A signature syntactically describes a message by specifying the message name and the types of its arguments and results. Commonly, a signature may also contain a list of signaled exceptions [SCB+86, KMN+90, DGLM95], but we will postpone this aspect until Section 2.2.6. A specific method satisfies a given signature if it has the required name and argument and result types.

When an object's behavior is reduced to the messages it can answer (described by signatures), the object's type specification is naturally reduced to an *interface*:

Interface: An interface specifies the behavior common to a set of objects in terms of the messages that can be sent to these objects. An interface consists of a set of signatures[4]. An object x implements a given interface S if all the signatures specified in S have corresponding methods in x.

Example 2.2: The interface *IntStack*[5].
The behavior of integer stacks can be expressed by the following interface:

```
interface IntStack
    push(int) returns nothing
    pop() returns int
    isEmpty() returns bool
end IntStack
```

Every object of type *IntStack* must at least have methods named *push*, *pop*, and *isEmpty* with appropriate argument and result types. We will formalize the notion of interfaces in Section 2.2.4.

Type checking algorithms normally define types in terms of interfaces rather than complete behavioral descriptions [WZ88]. The compiler is usually only expected to automatically check syntactic compatibility based on interfaces. Clearly, interfaces consisting of message names and types are weaker than complete semantic specifications. Interfaces and signatures are, in general, underspecified, because they elide the behavior of the corresponding methods [WZ88]. However, this compromise has proven successful in practice, in terms of expressiveness, user acceptance, and automated compiler support [CCHO89, DT92].

Interfaces can ensure certain correctness properties of programs. In the context of object-oriented programming, this means that message sends are safe and can be checked at compile-time, eliminating the possibility of run-time type errors [CCHO89]. Strongly-typed languages

[4] Interfaces are very similar to abstract types [BHJL86], which define a collection of operation signatures.

[5] Throughout our examples will we commonly refer to well-known data structures such as sets, bags, and stacks. However, we will intentionally *not* attempt to be consistent in the presentation of a particular structure. For example, the interface *IntStack* might have one set of methods in this example and a different set in another example. We try to emphasize that there is no such concept as *the* stack [BI94].

can provide such safety by associating a compile-time type with each variable and formal parameter in the program, restricting them to refer to objects that belong to the associated type [LSAS77, AL90]. Whether these types are explicitly provided by the programmer or inferred by the compiler is irrelevant for the present discussion. Nevertheless, we are inclined to favor explicit typing, because it enhances the readability of programs [PS92].

Constrained Variable: A variable a may be *constrained* by a type τ, meaning that at run-time a may only refer to values x that belong to the type τ. The values x must satisfy the predicate associated with type τ ($\Leftrightarrow \psi_\tau(x)$).

In particular, all assignments to a constrained variable must respect that variable's type constraint. Parameter passing has the same semantics as assignment and can be defined in terms of assignment [LSAS77]. In particular, the formal parameters of a function can equally be constrained, meaning that the actual arguments must be consistent with the type constraints of the corresponding formal parameters. Constrained variables and formal parameters are restricted to refer only to objects that understand all messages sent to them via the variables. Depending on a type's specificity level, this "safety" may range from syntactic to behavioral guarantees.

Let us briefly turn our attention to the notion of "type" in object-oriented programming languages such as Smalltalk-80 [GR83]. Such languages are often mistakenly termed "untyped" [CCHO89, KMN+90, Ghe91], since there are no type constraints on variables—it is syntactically legal for any object to be assigned to any variable. Nevertheless, the concept of type plays a vital role in the mind of every programmer, because he will make assumptions about the properties of the objects he is dealing with [Ame87]. Consequently, although not declared, objects and variables have *implicit* types [Jon92]:

- The type of an object is defined as the set of public methods supported by that object[6].
- The "type" of a variable (meaning that variable's type constraint) is correspondingly defined as the set of methods applied via that variable. In other words, the variable's implicit type constraint is defined as the set of messages sent to objects referenced by that variable. This type is frequently a "fictional" type, since it usually consists of only a subset of some existing object type [Höl93, Jon92].

Languages like Smalltalk-80 should more appropriately be termed *dynamically-typed*, because type checking is performed, but is merely postponed until run-time. On the other hand, truly untyped languages such as Assemblers (and C to some extent) have only one fundamental data structure, namely the cells of raw memory [CW85, Car91]. In this sense, the term "type error" is misplaced in a such an untyped language, since the notion of "type" is absent [DD85].

Even in dynamically-typed languages it is important to have a good understanding of types [Ame87], since the primary goal (just as in any other language) must still be the avoidance of run-time type errors. Possibly, "types" in dynamically-typed languages are even *more* important than in strongly-typed languages, since the entire responsibility for managing these implicit types is laid upon the programmer, whereas in strongly-typed languages the programmer is aided by the compiler[7].

[6]For example, in Smalltalk-80, a *protocol* consists of a simple set of message selectors [Cook92].

[7]To enhance the legibility of their code, Smalltalk programmers compensate for the lack of explicit types by "typing" all variables via names that are indicative of their implicit type, for example: `anArray` [LP91].

2.2 Subtyping

The notion of subtyping is present in many programming languages [AC93]. In object-oriented systems, types can be arranged in a *type hierarchy* [Lis87] composed of *subtypes* and *supertypes* which are related to one another by the *subtype relation*. Intuitively, objects belonging to a subtype provide all the behavior and functionality of supertype objects, except that subtype objects can potentially do more [Lis87]. In other words, subtype objects are expected to behave like supertype objects [KMN+90], meaning that an object of a subtype can be used without type error *as if* it were an object belonging to the supertype [Por92]. The subtyping rules of a sound type system should be able to guarantee this substitutability at compile-time [Sny86].

Example 2.3: The types *Person* and *Student* describe person and student objects, respectively. It is usually expected that the type *Student* should be a subtype of the type *Person*, because a student can usually do everything that a normal person can (and possibly more)[8]. A function expecting an argument of type *Person* should also work if the actual argument is an element of type *Student*.

2.2.1 Principle of Substitutability

As has been widely observed [Sny86, Har91, Por92, ASB94, KLMM94, Tai96, Roy98], the main idea associated with the concept of subtyping is the *principle of substitutability* [WZ88]:

Principle of Substitutability: Informally, a type σ is a *subtype* of another type τ (written as $\sigma \leqq \tau$) if an expression of type σ can be used *in any context* that expects an expression of type τ [Por92, BCC+96]. Values of the subtype σ must be *substitutable* for values of the supertype τ.

As elaborated in Section 2.1, types should inherently describe behavior. Therefore, subtyping should describe the relationships between the behavior of objects. However, due to the intricacies in the formal specification of the semantics of objects, only their syntactic aspects are captured by signatures and interfaces. The notion of subtyping can be defined in terms of compile-time checkable constraints on interfaces, or in terms of semantic restrictions [WZ88]. Based on this distinction (between types that describe syntactic and types that describe behavioral aspects of objects) one can distinguish between two different kinds of substitutability [Por92]:

Weak Principle of Substitutability: A type σ is a subtype of another type τ if expressions of type σ can be substituted into programs expecting expressions of type τ, while guaranteeing the absence of run-time type errors ("message not understood"). This principle guarantees *syntactic compatibility* between a supertype and its subtypes.

[8] Nevertheless, we reject generalizations such as "*Student* is a subtype of *Person*", since such a relation crucially depends on the particular definitions of *Student* and *Person* [BI94]. Similarly, stating that a *Set* is not a subtype of *Bag* is not generally valid, since these types can be defined to satisfy this relationship [Cook92].

Strong Principle of Substitutability: A type σ is a subtype of another type τ if for each object x of type σ there exists an object y of type τ such that, for all programs $\mathcal{P}$ defined in terms of type τ, the behavior of $\mathcal{P}$ is unchanged when x is substituted for y [Lis87]. Since subtype instances are constrained to have semantics consistent with their supertype, not only will there be no run-time type errors, but the original program behavior will remain unchanged. This principle[9] guarantees *semantic compatibility* between a supertype and its subtypes.

Since types describe externally observable behavior, subtyping is essentially a relationship on that behavior. In particular, just as type membership should be completely independent of specific object implementations, supertype and subtype objects may have different implementations as long as the corresponding behaviors can be correlated [LP91, BCC+96]. The relationship between concrete types and object implementations is dealt with in Chapter 3.

Example 2.4: Assuming that the type *IntStack* is a subtype of the type *IntBag*, objects of type *IntBag* may be implemented using arrays while objects of type *IntStack* may be implemented using linked lists. They have completely different internal representations but their types may nevertheless be related by subtyping.

2.2.2 Subtype Polymorphism

To define the meaning of subtyping more precisely, let us recall the definition of substitutability, namely that if σ is a subtype of τ ($\sigma \leqq \tau$), then values of σ must be able to substitute values of τ in any context. If the values of σ "already" have all the properties required by the predicate associated with τ, then the requirement of substitutability is trivially fulfilled, since then σ's values *are* also of type τ.

Instead of stating that a value belonging to a subtype can be used *as if* it were a supertype value, we will simply base our definition of subtyping on the premise that subtype values actually *are* supertype values [Ame90]:

Subtyping: A type σ is a subtype of another type τ ($\sigma \leqq \tau$) if any expression belonging to σ also belongs to τ [Ame87, Car91, Ghe91, BCC+96, Ame90]. In terms of predicates, type σ is a subtype of another type τ, with associated predicates ψ_σ and ψ_τ, respectively, if $\psi_\sigma(x) \Rightarrow \psi_\tau(x)$ for all values x. In other words, if the elements of σ satisfy the constraints imposed by ψ_τ, then $\sigma \leqq \tau$ [Sny86].

It should be noted that since predicates define sets of objects, types and subtypes correspond to sets and subsets, respectively [Sou97]. According to the definition of subtyping, the set defined by the predicate associated with a supertype includes all objects belonging to subtypes. Therefore, the subtype relationship is basically an inclusion relationship between types [AC93, Car91]—subtypes are subsets. Subtyping in [WZ88] is defined as predicate modification in terms of constraints that determine a subset of the set defined by the parent predicate.

[9] A long debate has ensued on whether behavioral subtyping is overly restrictive in practice, stringently precluding relationships which are commonly appreciated [LW90, DT92, LW94].

As mentioned earlier, our definition of types and subtyping coincides naturally with polymorphic type systems [CW85], in which an expression may have many, possibly incomparable types. In particular, our definition of subtyping automatically gives rise to a form of polymorphism intimately connected to subtyping and described in [Car88], namely *subtype polymorphism* (also known as *subsumption* [BCC+96] and *inclusion polymorphism* [CW85, MDCB91]):

Subtype Polymorphism: If an expression e belongs to a type σ, and another type τ is a supertype of σ, then e also has type τ. More formally, $e : \sigma \wedge \sigma \leqq \tau$ implies $e : \tau$. The phrase "type of an object" usually means the *most specific* such type.

The notion of subtype polymorphism has immediate impacts on the legality of assignments and method applications. The definition of constrained variables (Section 2.1), which determines the validity of assignments and applicability of methods, is implicitly extended to accommodate subtyping: if a variable a is restricted to values of a type τ, and type σ is a subtype of type τ, then a may naturally also refer to values of type σ. Due to subtype polymorphism, these values *are* also of type τ[10].

Consequently, in statically type-checked programming languages such as C++ [Str97] and Trellis/Owl [SCB+86], the subtyping rules are of critical importance since they determine the type correctness and hence the legality of programs [Sny86]:

1. In strongly-typed object-oriented languages the concept of subtype polymorphism is used to "broaden" the assignment statement [LW93]: the assignment of an object x to a variable constrained by the type τ is considered legal only if the type of x is a subtype of τ.

2. The same rules apply to formal parameters and actual arguments. The application of a function with formal parameters constrained by the types $\sigma_1, \ldots, \sigma_n$ to actual arguments with types $\tau_1, \ldots, \tau_n$ is considered legal if $\tau_i \leqq \sigma_i$ for all $i \in \{1, \ldots, n\}$. In other words, subtype objects can be passed to functions expecting supertype objects [LW90, BCC+96].

Variables are polymorphic in the sense that they may refer to values of subtypes, and functions are polymorphic in the sense that they may be applied to expressions of subtypes [CCHO89]. Although functions may have been defined with merely the supertype in mind, they are yet applicable to arguments of types that will only be defined in the future. This open-ended character is essential in large systems, permitting extensions to existing facilities without necessarily modifying them [Car88b].

To round off this section, we will briefly discuss some general subtype properties which are applicable to every subtype relation, independent of the particular kind of predicate logic in use. First, the subtype relation is typically a *partial order* on types [PS92], meaning that not all types are necessarily in a subtype relation. Further, the subtype relation is *reflexive* and *transitive* [LW90, Ghe91]:

[10] Some researchers [PS92, MMM90] propagate the necessity of *reverse assignments*, in which supertype values are assigned to variables constrained by subtypes. For example, the type information lost when inserting elements into a heterogeneous collection may need to be dynamically recovered. Such reverse assignments usually entail implicit run-time type checks, as is the case with the *assignment attempt* operation in Eiffel [Sal98] and with *dynamic casts* in C++ [Str97].

Reflexivity: Every type τ is a subtype of itself: $\tau \leqq \tau$. According to the subtyping definition, this requires $\psi_\tau(x) \Rightarrow \psi_\tau(x)$ for all values x, which is trivially true.

Transitivity: If $\sigma \leqq \rho$ and $\rho \leqq \tau$, then $\sigma \leqq \tau$. According to the subtype definition, this requires $((\psi_\sigma(x) \Rightarrow \psi_\rho(x)) \wedge (\psi_\rho(x) \Rightarrow \psi_\tau(x))) \Rightarrow (\psi_\sigma(x) \Rightarrow \psi_\tau(x))$ for all values x, which follows from the transitivity of implication.

Having explored the notion of subtyping at a rather abstract level, imposing no special restrictions on the predicates that describe the types, in the following sections we will turn our attention to more concrete levels of type specificity, namely to interfaces and object types which describe syntactic aspects of types and compatibility. Ultimately, we strive to describe the subtyping relation for object types based on the weak principle of substitutability, by subsequently specializing the general notion of subtyping for different type constructors: range types, function types, and object types. An attempt is made to present subtyping as an orthogonal concept which is independently applicable to all type constructors [Car88b].

2.2.3 Functions

Although the behavioral specification of functions is important, in this study we will concentrate on functions as specified by signatures (Section 2.1) in terms of function names and argument and result types. However, before turning to functions and function subtyping, we will introduce subtyping for ranges [CW85]. Range types and subtypes are simple enough to considerately introduce the mechanisms used throughout the later sections:

Range Type: A range type τ describes a range of elements between a lower bound n and an upper bound m (written as $\{n \dots m\}$). The associated predicate ψ_τ is defined as: $\psi_{\{n \dots m\}}(x) = n \leq x \leq m$, whereby $\leq$ denotes "less than or equal to" and not the subtype relationship.

Subtyping for Range Types: A range type $\{n \dots m\}$ is a subtype of another range type $\{n' \dots m'\}$ if $\psi_{\{n \dots m\}}(x) \Rightarrow \psi_{\{n' \dots m'\}}(x)$ for all elements x. From the following implication $(n \leq x \leq m) \Rightarrow (n' \leq x \leq m')$, we can deduce that $\{n \dots m\} \leqq \{n' \dots m'\}$ exactly if $n' \leq n$ and $m \leq m'$.

Example 2.5: Range types and subtypes.
The integer range type $\{4 \dots 6\}$ is a subtype of the integer range type $\{3 \dots 7\}$, since an integer between 4 and 6 can be used in any context that expects an integer between 3 and 7. However, the type $\{4 \dots 6\}$ is not a subtype of the type $\{5 \dots 7\}$, since the integer 4 may not be used in a context that expects an integer between 5 and 7.

For reasons that will soon become clear, we are especially interested in the typing issues raised when functions are passed as arguments or assigned to variables [Cas95]. In other words, we are interested in *higher-order* functions[11] [MHH91, CW85] and regard functions as first-class values [Cas95].

[11] Higher-order functions are functions that can take or return other functions as parameters [Car91, Cas95]. Consequently, functions themselves must be values with function types [Har91, MHH91].

Our interest in higher-order functions is founded in the fact that the object-oriented message passing paradigm inherently involves passing functions as arguments or assigning them to variables [Har91, Cas95]. Even though a programmer may not write higher-order functions consciously, when an object x is passed as an argument to a function f, then its methods are actually being passed too [Har91]. When f sends a message to x, then the inherently passed method is invoked—methods *are* higher-order functions.

Therefore, methods are subject to substitutability: assuming f works type-safely with x, then another object y can type-safely be substituted for x only if the methods of y are substitutable for the methods of x. In other words, y may substitute x if the messages (with the associated arguments) originally sent to x by f can equally well be handled by y.

We are concerned with methods being inherently passed as arguments that can substitute the expected methods without leading to type errors. This interaction between substitutability and higher-order functions leads to the notion of function subtyping. Function subtyping describes when a function can type-safely replace another function in every context. To start with, functions need to have *function types* [CW85], which are expressed in terms of their argument and result types [Har91]:

Function Type: A function type F, constructed from the argument type σ and the result type τ, is written as $\sigma \to \tau$[12]. The type F describes the set of functions that satisfy the associated predicate ψ_F, defined as follows: a function f has the type $\sigma \to \tau$, meaning it is an element of F ($\Leftrightarrow \psi_F(f)$), if f accepts *at least* all elements of type σ and returns elements of type τ. In other words, the *domain* of f must include the elements of σ and the *codomain* of f must be included in the elements of τ. If the sets $dom(f)$ and $codom(f)$ define the types ρ and ρ', respectively, then the predicate $\psi_{\sigma\to\tau}$ can be formalized using the subtype relation: $\psi_{\sigma\to\tau}(f) = (\sigma \leqq \rho) \wedge (\rho' \leqq \tau)$.

Function subtyping is concerned with the type-safe substitution of one function for another function [Har91]. In essence, the above definition already defines subtyping for function types. A function f with the domain type ρ and the codomain type ρ' inherently has the function type $\rho \to \rho'$. Therefore, showing that f also belongs to another function type $\sigma \to \tau$ is equivalent to showing that $\rho \to \rho'$ is a subtype of $\sigma \to \tau$. In other terms, a function f with the inherent function type $\rho \to \rho'$ (constructed from the f's domain and codomain) belongs to a function type $\sigma \to \tau$ exactly if $\rho \to \rho' \leqq \sigma \to \tau$. This can be formalized as:

Function Subtyping: A function type $\sigma' \to \tau'$ is a subtype of another function type $\sigma \to \tau$ if every function f of type $\sigma' \to \tau'$ also has type $\sigma \to \tau$. In terms of predicates, following the subtype definition (Section 2.2.2), we require $\psi_{\sigma'\to\tau'}(f) \Rightarrow \psi_{\sigma\to\tau}(f)$ for all functions f. Assuming ρ and ρ' to be the types corresponding to $dom(f)$ and $codom(f)$, respectively, substitution leads to $((\sigma' \leqq \rho) \wedge (\rho' \leqq \tau')) \Rightarrow ((\sigma \leqq \rho) \wedge (\rho' \leqq \tau))$, from which we can deduce that $\sigma \leqq \sigma'$ and $\tau' \leqq \tau$. To summarize, $\sigma' \to \tau' \leqq \sigma \to \tau$ exactly if $\sigma \leqq \sigma'$ and $\tau' \leqq \tau$.

The subtyping rules for function types can be intuitively explained as follows [BC96]: the subtype relation $\sigma' \to \tau' \leqq \sigma \to \tau$ holds if every function f of type $\sigma' \to \tau'$ can be used in

[12] Both the argument and the result type may themselves be constructed types, for example a function type $\sigma_1 \to \sigma_2$ or a cartesian product type $\sigma_1 \times \cdots \times \sigma_n$.

any context where a function g of type $\sigma \rightarrow \tau$ is expected. Since f's result of type τ' must be able to replace g's result of type τ, τ' must be a subtype of τ—values of type τ' are also of type τ. Similarly, since f must be able to handle any arguments that g can handle, meaning that f's domain must include at least g's domain, σ must be a subtype of σ'. Hence, f may safely substitute g without leading to run-time type errors.

Compared to the supertype, the subtype's domain expands while its codomain shrinks. The type constraints on the subtype's arguments are *less* stringent than those on the supertype's arguments. However, the type constraints on the subtype's results are *more* stringent than those on the supertype's results. This behavior is commonly characterized by calling the argument types *contravariant*[13], since they vary in the opposite direction of the subtype relationship, and the result types *covariant*, since they vary in the same direction as the subtype relationship [Har91, BC96, Pun99].

Example 2.6: Function subtyping and higher-order functions.
A function f with type $\{3, \ldots, 7\} \rightarrow \{7, \ldots, 9\}$ can be used in any context expecting a function with type $\{4, \ldots, 6\} \rightarrow \{6, \ldots, 10\}$, since it maps integers between 3 and 7 (and hence between 4 and 6) to integers between 7 and 9 (and hence between 6 and 10). The results will be in the expected range and at least the expected arguments can be handled. Thus, a function $h : (\{4, \ldots, 6\} \rightarrow \{6, \ldots, 10\}) \rightarrow \mathit{int}$[14] can be applied to the argument $f : \{3, \ldots, 7\} \rightarrow \{7, \ldots, 9\}$ without leading to run-time type errors.

It has been argued that function subtyping with contravariant argument types is counter-intuitive [Har91, DT92, PS92]. For example, assuming that *Employee* is a subtype of *Person*, then a function $f : \mathit{Employee} \rightarrow \mathit{int}$ should be more specific than a function $g : \mathit{Person} \rightarrow \mathit{int}$ [Har91]. As a matter of fact, the function f actually *is* more specific than g, in the sense that it operates only on a subset of the elements accepted by g (because the set of employees is a subset of the set of persons). Nevertheless, we have based our definition of subtyping on substitutability rather than on specificity. Clearly, f cannot replace g in contexts where the function is applied to objects of type *Person*. In this sense, it is perfectly intuitive that the function type $\mathit{Employee} \rightarrow \mathit{int}$ is not a subtype of $\mathit{Person} \rightarrow \mathit{int}$.

2.2.4 Object Types

Due to the fact that objects are central to the object-oriented paradigm, and that syntactic specification of objects satisfying the weak principle of substitutability is widely spread in the domain of practical programming languages [SCB+86, Ame87, Str97, GJS96], we will now explore *object types*, which describe syntactic issues of objects, and their subtypes. Lacking the formal specification of object behavior, we will deduce subtyping based solely on syntactic issues—the names and function types of an object's public operations [Sny86].

In Section 2.1 we informally introduce the notion of interface, which describes the behavior common to a set of objects in terms of the messages that can be sent to these objects. Interfaces are used throughout the entire presentation to illustrate our examples, attempting to achieve a fashion near to existing programming languages [Ada83, DGLM95, GJS96, BR97] and familiar to everyday programmers. To elegantly explain subtyping for objects in terms of

[13] The term *antimonotonic* is used in [AC93].
[14] This function takes a function argument and returns an integer result.

predicates, we need a concept more formal than interfaces, namely object types, which will be introduced in this section. However, let us keep in mind that interfaces and object types are mostly interchangeable, since they essentially describe the same thing, one a bit more formal than the other.

Interfaces contain only signatures, and signatures can be expressed as labeled function types, because they only contain names and argument and result types. Therefore, we may formalize interfaces as *object types*, which are sets of distinctly labeled function types[15]. This approach is similar to record types in [Car88, CHC90] and to object types in [BCC+96]. Objects can be modeled as record[16] values whose fields represent methods, and whose labels are the corresponding messages that invoke the methods [CP89, Cas95].

Object Type: An object type σ, constructed from the pairwise distinct labels $a_1, \ldots, a_n$ and the function types $\tau_1, \ldots, \tau_n$, is written as $\langle\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle\rangle$. The type σ describes the set of objects that satisfy the predicate ψ_σ, defined as follows: an object x has the type $\langle\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle\rangle$, meaning it is an element of σ ($\Leftrightarrow \psi_\sigma(x)$), if x has methods named $a_1, \ldots, a_n$ with corresponding function types $\tau_1, \ldots, \tau_n$. The predicate ψ_σ can be formalized as[17]: $\psi_{\langle\langle a_1:\tau_1,\ldots,a_n:\tau_n\rangle\rangle}(x) = a_1 : \tau_1, \ldots, a_n : \tau_n \ in\ x$.

Example 2.7: The interface *IntSet*.
Consider the following interface:

> `interface` *IntSet*
>   *ins*(*int*) `returns nothing`
>   *del*(*int*) `returns nothing`
>   *mem*(*int*) `returns` *bool*
>   *size*() `returns` *int*
> `end` *IntSet*

The interfaces *IntStack* (from Section 2.1) and *IntSet* correspond to the following object types:

$$IntStack \equiv \langle\langle push : int \to unit; pop : unit \to int; isEmpty : unit \to bool \rangle\rangle$$
$$IntSet \equiv \langle\langle ins : int \to unit; del : int \to unit; member : int \to bool; size : unit \to int \rangle\rangle$$

Every object of type *IntSet* must at least have methods named *ins*, *del*, *member*, and *size* with function types $int \to unit$, $int \to unit$, $int \to bool$, and $unit \to int$, respectively.

An object belongs to an object type if, for each message in the type, the object has a same-named method with a compatible signature [Por92]. Apparently, objects belonging to an object type σ may actually have *more* methods than specified by their type σ. The object type specification merely demands the existence of a particular set of methods with specific function types, but does not state that these are the *only* methods that σ's objects may have. This is intuitively appealing, since a context expecting an object of, for example, type *IntSet*

[15] Following [BCC+96], we use the type *unit* to denote the argument type of parameterless functions or the result type of procedures.

[16] A record is an unordered and finite association of values to labels. Various record-based object models have been proposed [Car88, CHC90].

[17] We assume that $a_1 : \tau_1, \ldots, a_n : \tau_n \ in\ x$ stands for: the object x has the public methods named $a_1, \ldots, a_n$ with function types $\tau_1, \ldots, \tau_n$, respectively. This informal definition is sufficient to derive the subtyping relationship between object types.

will at most send the messages named by *IntSet*. Therefore, it is type-safe for the object being manipulated to actually have other methods as well [BW86].

Although an object type σ requires its objects to have methods with specific function types, it is perfectly type-safe for the objects to have these methods with types that are actually *subtypes* of the required function types. Due to subtype polymorphism, these methods with function subtypes *are* also of the required function supertype, and can therefore replace them in any context without leading to run-time type errors [Cas95]. This is also known as *deep subtyping*, where a subtype's method can have a subtype of the corresponding method in the supertype. On the contrary, *shallow subtyping* requires that the common methods of the subtype and supertype have identical argument and result types [BP93]. Similarly to functions (Section 2.2.3), an object x with the "inherent" object type σ, constructed from the function types of the public methods of x, belongs to a type τ exactly if $\sigma \leqq \tau$:

Subtyping Object Types: An object type $\sigma \equiv \langle\langle a_1 : \sigma_1, \ldots, a_m : \sigma_m\rangle\rangle$ is a subtype of another object type $\tau \equiv \langle\langle b_1 : \tau_1, \ldots, b_n : \tau_n\rangle\rangle$ if every object x of type σ also has type τ. In terms of predicates, we require $\psi_{\langle\langle a_1:\sigma_1,\ldots,a_m:\sigma_m\rangle\rangle}(x) \Rightarrow \psi_{\langle\langle b_1:\tau_1,\ldots,b_n:\tau_n\rangle\rangle}(x)$ for all objects x. From $(a_1 : \sigma_1, \ldots, a_m : \sigma_m \textit{ in } x) \Rightarrow (b_1 : \tau_1, \ldots, b_n : \tau_n \textit{ in } x)$ we can informally deduce:

- $\{b_1, \ldots, b_n\} \subseteq \{a_1, \ldots, a_m\}$ $(n \leq m)$. This means that the subtype σ has *at least* all the messages of the supertype τ, namely $b_1, \ldots, b_n$, and possibly more. The labels of τ are included in those of σ.
- For all $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$ with $a_i = b_j$ we require $\sigma_i \leqq \tau_j$. For all message names that σ and τ have in common, the function type in σ must be a subtype of the function type in τ. In other words, σ's methods may have subtypes of the corresponding methods of τ. Therefore, following the contravariant nature of function subtyping, for every common method m, the argument types of m in type σ must be supertypes of the corresponding argument types in type τ, while the result type of m in type σ must be a subtype of the result type in τ.

Objects belonging to a subtype are more specific than the supertype's objects, in the sense that they may have additional operations and the methods they have in common are more specific in the subtype's objects [Har91]. The object type σ is a subtype of an object type τ if σ has at least the methods of τ (possibly more), and the function types of their common methods are related by the subtype relation [Cas95]. This is usually formalized as [BCC+96]: $\langle\langle a_1 : \sigma_1, \ldots, a_n : \sigma_n, \ldots, a_m : \sigma_m\rangle\rangle \leqq \langle\langle a_1 : \tau_1, \ldots, a_n : \tau_n\rangle\rangle$ $(n \leq m)$ exactly if $\sigma_i \leqq \tau_i$[18] for all $i \in \{1, \ldots, n\}$, whereby $a_1, \ldots, a_n$ are the common messages.

The terms "horizontal extension" and "vertical modification" are used in [WZ88] and [DT92] to denote the addition of new methods and the restriction of common methods, respectively:

[18] In record-based object models, which allow the record fields to represent public variables, a distinction must be made between mutable and immutable record fields. Mutable fields representing public variables may not be specialized in subtypes (they are *invariant*) because they can be read *and* written [Car88]. Specializing public variables leads to subtypes that are *read-only substitutable* for their supertypes [WZ88, DT92].

Horizontal extension: The subtype's objects can have a richer set of methods than the supertype objects. This increasing richness of methods decreases the number of objects possessing these methods. In effect, the set of objects belonging to the subtype is a subset of the set of objects belonging to the supertype.

Vertical modification: The subtype's objects have methods which may be function subtypes of the corresponding supertype methods.

Unfortunately, the expressions "extension" and "modification" suggest another sort of relationship between a supertype and its subtypes, namely a relationship where the subtype *shares* the messages of its supertype and *extends* or *modifies* the shared attributes. Therefore, we reject this terminology in connection with the subtype relationship and confine its application to another domain discussed in Chapter 4.

Example 2.8: Subtype may have more methods.
Bag and bounded bag objects are defined by the following object types:

$$\begin{aligned}&Bag \equiv \langle\langle put : int \rightarrow unit; get : unit \rightarrow int\rangle\rangle\\&BBag \equiv \langle\langle put : int \rightarrow unit; get : unit \rightarrow int; full : unit \rightarrow bool; size : unit \rightarrow int\rangle\rangle\end{aligned}$$

Type $BBag$ is a subtype of type Bag. Objects of type $BBag$ can be used without type error in any context that expects objects of type Bag, even though $BBag$'s objects have more methods.

Example 2.9: Deep subtyping.
Building on the previously defined object types Bag and $BBag$ we define:

$$\begin{aligned}&A \equiv \langle\langle foo : Bag \rightarrow unit\rangle\rangle\\&B \equiv \langle\langle foo : BBag \rightarrow unit\rangle\rangle\end{aligned}$$

Because of contravariance, the type A is a subtype of type B and not the other way round. Objects of type A can be used without type error in any context that expects objects of type B, even though the function type of foo in A is a subtype of foo in B.

Some languages such as C++ [Str97] and Java[19] [GJS96] don't support function subtyping at all, although they might support higher-order functions and function types [Har91, Sha94]. Consequently, the argument and result types are *invariant*:

- An object type σ is a subtype of another object type τ if σ has at least the methods of τ and their common methods have *identical* function types.
- When a function value is assigned to a variable or passed as a parameter, then its function type must be *identical* to the function type constraining the variable or formal parameter.

[19] Language specifications prior to [GJS96] were more general in that they allowed covariant result types. In contrast, [GJS96] requires the result types to be invariant [DE97].

2.2.5 Structural vs. Named Subtyping

Our present definition of subtyping for object types is based on *structural equivalence*, where the declared name of a type has no influence on whether it is equivalent to or a subtype of another type [KLMM94]. This approach is very common in type-theoretic research [Car88, AC93, BCC+96]. In a sense, an object type name σ is merely an *abbreviation* for the corresponding type expression $\langle\langle a_1 : \sigma_1, \ldots, a_n : \sigma_n \rangle\rangle$. For example, the name Bag is an abbreviation for the type expression $\langle\langle put : int \to unit; get : unit \to int \rangle\rangle$. Basically, two type expressions are considered equivalent if they have the same structure, regardless of the names of the abbreviations [CW85, AC93]. Basic types are structurally equivalent to themselves [Car88].

Structural Subtyping: The subtype relationship is based on structural type equivalence. The underlying compiler *infers* whether two object types are in a subtype relationship, based on their type structures and completely irrespective of the particular names involved.

Example 2.10: Structural equivalence and subtyping.
Assume the following object types:

$$
\begin{aligned}
A &\equiv \langle\langle x : int \to int \rangle\rangle \\
B &\equiv \langle\langle x : int \to int \rangle\rangle \\
C &\equiv \langle\langle y : A \to A \rangle\rangle \\
D &\equiv \langle\langle y : B \to B; z : int \to int \rangle\rangle
\end{aligned}
$$

The object type A is structurally equivalent to the object type B. Each is a subtype of the other: $A \leqq B$ and $B \leqq A$. The object type D is a subtype of the object type C, since D has a method y, and the associated function type $B \to B$ is a subtype of the function type $A \to A$. However, because D has an extra method $z : int \to int$, the opposite is not true: C is not a subtype of D.

A term frequently used synonymously with structural subtyping, especially in the context of abstract data types [LSAS77, BHJL86] and interfaces [PS92], is *conformance* [Horn87, BGM89, KMN+90]. We will retain this terminology, using *conformance* to denote structural subtyping, while the more general term subtyping encompasses both structural and named subtyping, which is discussed below. Conformance is commonly the basis for compile-time type checking of the legality of assignments: if the variable x is statically declared with type σ, then an object of type τ may be safely assigned to x if τ conforms to σ [BHJL86, BGM89, KMN+90]. This guarantees that objects assigned to x implement at least the methods defined by the type constraint σ.

Purely structural equivalence and subtyping are not very popular in practice [Höl93]. Most commercially available and widely used object-oriented programming languages such as Ada [Ada83], C++ [Str97], Java [GJS96], and Eiffel [Mey97] are based mainly on *name equivalence* and on *named subtyping* [KLMM94]. With name equivalence, each type specification defines a unique type that is not equal to the type defined by any other type specification [DGLM95]. Type equivalence in such systems is determined by the names assigned to the types, rather than by their structures [AC93].

Named Subtyping: The subtype hierarchy among user-defined types is declared *explicitly*: a subtype specification *names* the type's direct supertype(s). A type τ is a *direct* supertype of another type σ if this is declared in σ's specification *and* the declaration is legal, in the sense that σ conforms to τ.

Clearly, it would be an irreconcilable hindrance if *all* subtype relations had to be stated explicitly. In particular, relationships adhering to the transitive and reflexive nature of the subtype relation are still inferred by the compiler [PS92]: a type σ is a subtype of a type τ if either σ and τ are identical, or if σ is a direct subtype of some other type ρ, and ρ is a subtype of τ [Por92].

Example 2.11: Named subtyping.
Consider the following type specifications (borrowed from [ASB94]):

```
interface SimpleStack
    push(int) returns nothing
    pop() returns int
end SimpleStack

interface Stack subtype of SimpleStack
    push(int) returns nothing
    pop() returns int
    swaptop() returns nothing
end Stack
```

The type *Stack* is declared to be a subtype of *SimpleStack*. Since the corresponding object types are in the correct conformance relationship, the declaration is legal[20].

There are a number of various different reasons explaining the lack of popularity of structural subtyping in practice, ranging from the loss of abstraction to the accidental conformance of unrelated types:

- Structural subtyping is inherently entailed in a loss of abstraction [Höl93]. With name equivalence, types describing different behaviors, but with identical or conforming type structures, can nevertheless be distinguished by their names. Thus, the type's name is essentially part of the behavioral description. Consequently, declared subtype relationships provide a finer granularity of type checking than is possible when type relations are inferred [DGLM95].

- Structural subtyping is less transparent than explicitly named subtyping [BP93]. With named subtyping, the whole subtype hierarchy is more explicit and easier to understand. Subtypes should be made with subtyping in mind [LW90], rather than being the "coincidental" product of a clever compiler.

[20] It is important to note that, although *Stack* is explicitly declared to be a subtype of *SimpleStack*, the *complete* interface of *Stack* is specified. *Stack* does not *inherit* anything from *SimpleStack* in any sense, but simply states a subtype relationship satisfying the weak principle of substitutability. We will discuss the issue of *interface inheritance* in greater detail later. At this point, we merely emphasize that subtyping and interface inheritance are fully independent mechanisms.

- Structural subtyping naturally facilitates multiple subtyping, where a type may be a subtype of several types—conflicts cannot arise by definition [Ame87]. However, a type cannot be a subtype of another type with different message names. Declared subtyping allows the supertype's messages to be projected onto the subtype's messages by means of renaming [DGLM95].

- Structural subtyping can lead to accidental conformance, in the sense that the compiler may establish unintended subtype relations between conceptually unrelated types [PS92, Höl93]. Types not related in the application domain may be related in the program simply as the result of coincidence [DT88].

Example 2.12: Coincidental conformance.
Assume the following object types (similar to [PS92, Tho97]):

$$GraphicalObject \equiv \langle\langle draw : unit \rightarrow unit \rangle\rangle$$
$$Cowboy \equiv \langle\langle draw : unit \rightarrow unit ; shoot : unit \rightarrow unit \rangle\rangle$$

When structural subtyping is employed, a function f with type $GraphicalObject \rightarrow unit$ is "coincidentally" applicable to arguments of type *Cowboy*, because *Cowboy* happens to conform to *GraphicalObject* [Höl93, Tho97]. Clearly, passing a cowboy to a function expecting a graphical object is most probably not the function's foreseeable area of applicability.

Example 2.13: Conformance reduces abstraction.
Consider the following interfaces:

```
interface Set
    put(int) returns nothing
    get() returns int
end Set

interface Bag
    put(int) returns nothing
    get() returns int
end Bag
```

With name equivalence, the types *Set* and *Bag* are unique and distinct types, even though their type structures are identical—the types are not in any subtype relationship. This distinction is intuitively appealing, since *Set* and *Bag* are different types and not behaviorally substitutable for each other[21]. In the case of structural equivalence and subtyping, the types *Set* and *Bag* are equivalent. Each is a subtype of the other [DGLM95], meaning that objects of *Set* can be substituted for objects of *Bag*, and vice versa. Consequently, we have lost an important opportunity for type checking.

As already mentioned, structural subtyping is very popular in type-theoretic research [CW85, Car88, Car88b] and in exploratory programming [BHJL86, AL90, KLMM94, BR97]. Structural subtyping does indeed have certain advantages over named subtyping:

[21] A context expecting a bag will probably not perform as desired if handed a set, because the context may rely on the existence of duplicates. However, [Cook92] shows that, with appropriate definitions, a set *can* be behaviorally substitutable for a bag.

- Structural subtyping is more flexible than named subtyping [BP93]. Since the subtype relations are not declared explicitly, but inferred by the system, all possible subtype relations implicitly exist. Retrofitting a common supertype (a frequent problem in systems with named subtyping [Höl93, BR97]) is simply accomplished by defining that type: the relationships to its subtypes will be derived by the compiler.

- Structural subtyping gives the software designer and prototyper a flexible tool when progressing through the development of new software [KLMM94], and avoids splitting the type hierarchy into every possible opportunity for code reuse [Höl93].

- Structural subtyping may be the only possible mechanism to realize a certain concept. In particular, the notion of *f-bounded parametric polymorphism* [CCHO89, CHC90] can only be reasonably established by means of conformance. The same applies to similar mechanisms such as *with* clauses in Ada [Ada83] or *where* clauses in Theta [DGLM95] and CLU [LSAS77].

- Since purely named subtyping is never used in languages, it is important to study structural subtyping to understand the reconciliation of structural and named subtyping in a single system [Car88b].

Example 2.14: Structural subtyping is more flexible.
The following interface represents objects that have a *print* method, possibly returning some string representation of themselves:

```
interface Printable
    print() returns string
end Printable
```

Assume a function $display : Printable \rightarrow unit$ that restricts its arguments to subtypes of *Printable*. Using structural subtyping, any type with a method $print : unit \rightarrow string$ conforms to *Printable*, meaning that its objects can be passed to *display*. With named subtyping, this potential reuse must be discovered when designing the type hierarchy and *a priori* to defining *display*. Any type whose objects are to be passed to *display* must explicitly state its subtype relationship to *Printable*. Clearly, the latter approach leads to a proliferation of possibly useless types and subtype relations.

Ideally, one should strive for a hybrid approach, picking the best from both worlds. We believe that it would bear unsurpassable advantages to reconcile structural and named subtyping and make them both available in a single type system. However, to enhance the readability and maintainability of programs, the long-term goal should nevertheless be to have all subtype relations explicitly declared [PS92]. For example, subtyping could be structural by default, but the programmer would have the possibility to additionally state explicit subtype relations, which would then be checked by the compiler [AL90]:

- In the early phases of design, structural subtyping would support fast prototyping, providing some of the flexibility of dynamically-typed languages [Car88b, Höl93] while retaining the security of full compile-time type checking [KLMM94].

- Later, when the subtype relations, which always play an important role in the mind of every programmer [Ame87], regardless of whether subtyping is structural or named, have crystallized, then they can be "fixed" by declaring them explicitly. Once a type's supertypes have been declared, this type is excluded from the conformance relationship, meaning that its *only* direct supertypes are those explicitly declared[22].

Thus, the indisputable benefits of named subtyping would prevail, drastically improving the legibility of programs [PS92], refining the granularity of type checking [DGLM95], and unmasking coincidental and unintended subtype relationships [PS92, Höl93].

2.2.6 Interface Containment

In a more practical vein, to clarify the issues of subtyping for function and object types, we will informally apply our previous results to interfaces and signatures as introduced in Section 2.1. In the context of interfaces, the mechanism analogous to subtyping (and conformance in particular) is the notion of *interface containment* [CCHO89, Har91]. An interface S is said to *contain* another interface T if their corresponding object types are in a conformance relationship:

Interface containment: Each interface contains itself. An interface S contains another interface T if for each signature m_T in T there exists a corresponding signature m_S in S, such that:

- m_T and m_S have the same name and the same number of arguments and results. Assume that m_S takes arguments with interfaces $\alpha_1, \ldots, \alpha_n$ and returns a result with interface γ, while m_T takes arguments with interfaces $\beta_1, \ldots, \beta_n$ and returns a result with interface δ. The corresponding function types of m_S and m_T are $\alpha_1 \times \cdots \times \alpha_n \to \gamma$ and $\beta_1 \times \cdots \times \beta_n \to \delta$, respectively.
- *Contravariance of arguments.* Each argument interface of m_T contains the corresponding argument interface of m_S: β_i contains α_i for all $i \in \{1, \ldots, n\}$.
- *Covariance of results.* If m_S and m_T have a result, then the result interface of m_S contains the result type of m_T: γ contains δ.
- *Covariance of exceptions.* If signatures include exceptions [SCB+86, KMN+90], we require that the exceptions listed by m_S are contained in the set of exceptions listed by m_T[23] [DGLM95, LW93, Tho97].

This is the usual definition of interface containment [CCHO89] extended to support exceptions [DGLM95]. The interface S provides at least the signatures of T (S may have more signatures), and the signatures that S and T share in common have contravariant arguments and covariant results.

[22]Subtype relations resulting from transitivity may still be inferred according to the rule stated earlier, but using only the type's declared direct supertypes. Therefore, a type σ that is declared to be a subtype of a type τ can *only* also be a subtype of another type ρ if τ is a subtype of ρ.

[23]If the exceptions themselves are related in a subtype hierarchy, then m_S may list exceptions that are subtypes of the exceptions listed by m_T [MT97].

The contravariant argument and covariant result rules ensure that compile-time type checking can guarantee the absence of run-time type errors [DGLM95]: every object x of type S will behave like an object of type T, with respect to the messages x can accept: x will accept each message listed in T, can handle all arguments with interfaces $\beta_1, \ldots, \beta_n$, and will return a result with interface δ [Ame87]. In fact, x can handle arguments with the more general interfaces $\alpha_1, \ldots, \alpha_n$ and will return a result with the more specific interface γ.

The rule for exceptions ensures that a signature in S lists only exceptions defined for the corresponding signature in T, meaning that m_S may not signal more exceptions than m_T. Thus, a caller of m using the object through the interface T will receive only exceptions declared in m_T. The caller should not be expected to handle exceptions not listed in the interface being used[24] [LW93].

2.3 Recursive Types

Types may be defined *recursively*, referring to themselves in their respective specifications. In particular, object types and interfaces can have recursive (and therefore potentially infinite) structures [CCHO89]:

Recursive Type: An object type $\sigma \equiv \langle\langle a_1 : \tau_1, \ldots, a_n : \tau_n \rangle\rangle$ is recursive if σ appears as the argument type or as the result type in one of the function types $\tau_1, \ldots, \tau_n$. An interface T is recursive if T appears as the argument interface or as the result interface in one of T's signatures.

Recursive types and interfaces are no rarity. In fact, they are very common in practice [Ame90, AC93, BCC+96, Pun99], being just as fundamental and immanent to object-oriented programming as recursive functions are to procedural programming[25] [WZ88, CP89].

Example 2.15: Recursive interfaces and object types.
For example, consider the interface

$$
\begin{array}{l}
\texttt{interface}\ Bag \\
\quad put(int)\ \texttt{returns}\ Bag \\
\quad get()\ \texttt{returns}\ int \\
\quad equal(Bag)\ \texttt{returns}\ bool \\
\quad union(Bag)\ \texttt{returns}\ Bag \\
\texttt{end}\ Bag
\end{array}
$$

with the corresponding object type

$$Bag \equiv \langle\langle put : int \rightarrow Bag; get : unit \rightarrow int; equal : Bag \rightarrow bool; union : Bag \rightarrow Bag \rangle\rangle$$

The interface Bag appears as the result interface of put, as the argument interface of $equal$, and as the argument and result interface of $union$. The recursive occurrence of the name Bag in the object type Bag is recursively bound to the object type itself.

[24] A recently proposed view [MT97] suggests that exceptions may need to be non-conform, since exceptions are in part dependent of particular implementations.

[25] We are mainly concerned with *directly* recursive types. Examples of mutually (indirectly) recursive types are given in [Tho97, BOW98].

Due to the popularity of recursion in interfaces, it is important to understand the interplay between recursive object types and subtyping. In Chapter 6 we will see how recursive types lead to serious typing problems in object-oriented programming.

Subtyping Recursive Types: A recursive object type σ is a subtype of another recursive object type τ if, *assuming* that type σ *is* a subtype of type τ, the recursion-free version of type σ is a subtype of the recursion-free version of type τ[26] [CCHO89].

Example 2.16: Recursive subtypes.
The following object types are recursive:

$$A \equiv \langle\langle f : int \rightarrow A \rangle\rangle$$
$$B \equiv \langle\langle f : int \rightarrow B ; x : unit \rightarrow int \rangle\rangle$$

To show that the recursive type B is a subtype of the recursive type A, we simply assume $B \leqq A$ and show that $\langle\langle f : int \rightarrow B ; x : unit \rightarrow int \rangle\rangle \leqq \langle\langle f : int \rightarrow A \rangle\rangle$. The subtype relation must hold between the recursion-free versions where A and B are not tied to any type expression. This is apparently true due to the covariant rule for result types. Note that if B did not have the extra method $x : unit \rightarrow int$, then the subtype relation would also hold in the opposite direction, namely $A \leqq B$.

Example 2.17:
Consider the following object types:

$$A \equiv \langle\langle f : A \rightarrow int \rangle\rangle$$
$$B \equiv \langle\langle f : B \rightarrow int ; x : unit \rightarrow int \rangle\rangle$$

To show that the recursive type B is a subtype of the recursive type A, we would have to show that $\langle\langle f : B \rightarrow int ; x : unit \rightarrow int \rangle\rangle \leqq \langle\langle f : A \rightarrow int \rangle\rangle$ where A and B are free. However, assuming $B \leqq A$, the subtype relation does not hold, since the contravariant rule for argument types requires exactly the opposite of our assumption, namely $A \leqq B$.

Example 2.18: *Point* and *ColorPoint*.
A classical example found in literature [BCC+96, BC96, CCHO89, MHH91] is typically depicted by the following object types:

$$Point \equiv \langle\langle x : unit \rightarrow real ; y : unit \rightarrow real ; equal : Point \rightarrow bool \rangle\rangle$$
$$\begin{aligned} ColorPoint \equiv \langle\langle & x : unit \rightarrow real ; y : unit \rightarrow real ; color : unit \rightarrow string ; \\ & equal : ColorPoint \rightarrow bool \rangle\rangle \end{aligned}$$

Intuitively, *Point* and *ColorPoint* appear to be in a subtype relationship. However, they are not, even though they have similar recursive structures. The contravariant rule for argument types requires the parameter of *equal* in *ColorPoint* to be a supertype of *Point*, and not a subtype as assumed. In Chapter 6 we will return to this example in more depth, exploring ways to regain the lost subtype relationship. In Section 4.4 we will describe how *ColorPoint* can be incrementally defined in terms of *Point*, exploiting the similarities they share in common.

[26] In the recursion-free version of a recursive object type τ, the recursive reference of τ to itself is not tied to the type but is free.

Chapter 3

Implementations

In the previous sections we frequently mentioned concrete objects and how their behavior satisfies specific object types. The introduction stated that objects have some internal state and methods, implementing the externally observable behavior. However, we deferred a more detailed discussion of implementations to a later point in time. This section describes object implementations and their relationship to object types[1].

3.1 Object Implementations

As elaborated in Section 2.1, types classify objects according to their externally observable behavior. Similarly, now that we are primarily interested in possible realizations of this behavior, objects can be classified according to their internal structure and algorithms—objects can be grouped into *implementations* [Por92]:

Implementation: An implementation classifies a set of objects according to their common, internal implementation details [Ame87, PS92, Sou97]. An implementation is typically characterized by a name, a set of variable declarations, and a set of method definitions [Cas95]. An object x is called an *instance* of an implementation I if x has the internal structure and algorithms specified by the description of I [BCC+96].

In normal object-oriented jargon the term "class" is used instead of "implementation", in the sense that objects are grouped into classes and each object is an instance of some class. Unfortunately, the term "class" has adopted different meanings in the course of time. For example, in POOL-I [AL90] classes describe purely implementational issues independent of types, while in C++ [Str97] and Eiffel [Mey97] classes are special combinations of types and implementations (described in Section 5.1.1). To emphasize this distinction we will use the term "implementation" to denote the former and "class" to denote the latter.

An implementation specifies a representation schema and a set of method bodies for a collection of objects [LSAS77, Por92]. All objects of the same implementation have exactly the same structure of their internal data. In particular, they have the same set of variables[2],

[1]Many of the issues treated in this chapter are applicable to prototype-based languages as in [Lie86, US87].

[2]Clearly, each object has its own private set of these instance variables with different values, but the names and type constraints of the variables are the same along all instances of an implementation [Ame90].

representing their internal state, and the same set of method bodies. In general, an implementation may differentiate between *public* and *private* methods. The former are visible to users of the instances, while the latter may only be used internally by other methods. To enforce encapsulation, we assume that instance variables are invisible to external users and are only accessible from within an object's methods.

In a sense, an implementation is a template (cookie cutter) from which objects are created [WZ88, Tai96], describing how its instances are built [AL90]. Some object-oriented programming languages, for example Smalltalk-80 [GR83], view implementations themselves as objects, with methods to create new instances [Ame87, DT88]. However, for the sake of simplicity in illustrating our examples, we choose a different approach taken in [CCHO89, BCC+96] and similar to C++ [Str97]: the primitive `new` is used to generate instances of an implementation from the implementation's name, possibly taking a set of initialization arguments which are assigned to the corresponding instance variables. An implementation definition should provide exactly the information necessary to create new objects [Ame90].

Example 3.1: The implementation *Person*.
The following implementation describes person objects:

```
implementation Person(name : string, age : string)
    var name : string := name
    var age : string := age
    method name() returns string
        return name
    method age() returns int
        return age
    method happyBirthday() returns nothing
        age := age + 1
end Person
```

Instances of *Person* have the internal state represented by the instance variables *name* and *age*, both of which are constrained by the type *string*. However, this internal state is not visible to users of person objects. Instead, the name and the age of a person can be queried by sending the messages *name* and *age* to person objects[3]. In addition, persons accept the message *happyBirthday*, which increases the person's age. The method bodies in the definition of *Person* describe how to react to these messages.

Example 3.2: The implementation *Rectangle*.
Consider the following implementation of a rectangle:

```
implementation Rectangle(x : int, y : int)
    var x : int := x
    var y : int := y
    method width() returns int
        return x
    method height() returns int
        return y
```

[3]Since the syntax of variable access and message sending is distinct, the same name may be used to denote both a variable and a method.

```
        method area() returns int
            return self.width() * self.height()
        method equal(r : Rectangle) returns bool
            return self.width() = r.width() and self.height() = r.height()
    end Rectangle
```

Instances of *Rectangle* have an internal state represented by the instance variables x and y. The height and the width of a rectangle can be queried by sending the messages *height* and *width*. The area of a rectangle can be computed or the rectangle can be compared to another rectangle with the messages *area* and *equal*[4], respectively. The implementation definition provides the corresponding methods.

The definition of *Rectangle* exemplifies a commonly encountered characteristic of object implementations—their recursive structure [CCHO89]. The implicitly declared pseudo-variable[5] `self` in a method body indicates such recursive references. The variable `self` denotes the receiver of the message, meaning the object on whose behalf the current method is being executed. When a method is invoked, `self` is bound to the receiver of the message, allowing the object to send messages to itself via `self`, or even to pass itself as a message argument to other objects.

3.2 Types and Implementations

An implementation provides the realization of some object behavior [LP91]. We are interested in the *type* of the instances produced by an implementation. In other words, given a specific implementation, we would like to classify the behavior of its instances. Since types should describe externally observable behavior only, we may immediately disregard an object's instance variables and private methods, because both are invisible and inaccessible to the object's users. Only the object's public methods constitute its behavior as seen by "outsiders". Therefore, as previously hinted in Section 2.2.4, we may associate with each object an object type, constructed from all the function types of the object's public methods.

Example 3.3: If a given object x has the public method $plok : string \rightarrow int$, then, according to the definition of object types, x satisfies the predicate associated with the object type $\langle\langle plok : string \rightarrow int\rangle\rangle$.

If one particular object belongs to a certain type, then all instances of that object's implementation must also belong to this type, because all instances of the implementation have the same immutable properties [Ame90]. Since the behavior of an object is ultimately determined by its internal structure and algorithms [LP91], and all instances of an implementation have the same internal structure and algorithms, an implementation's instances all exhibit

[4] The type constraint of the formal parameter r denotes the *object type* associated with the implementation *Rectangle* rather than the implementation itself. Such automatically associated types are explained below.

[5] Special references, managed internally by the system, can be made accessible to the programmer in form of *pseudo-variables* [Tai96]. A pseudo-variable is not a real variable—it cannot be changed or reassigned by the programmer. Programming languages usually offer pseudo-variables such as `self` and `super`.

the same uniform behavior [Weg87], and belong to the same type. We can automatically associate with every implementation an object type with the same name[6] [AL90, Ame90]:

Implementation defines Type: An implementation I automatically defines the associated object type I. All instances of I belong to the object type I. This object type is constructed by collecting all the function types corresponding to the public methods specified in the definition of I.

Example 3.4:
The previously specified implementations *Person* and *Rectangle* can be associated with the following object types:

$$\begin{aligned}
&Person \equiv \langle\langle name : unit \to string; age : unit \to int; happyBirthday : unit \to unit\rangle\rangle \\
&Rectangle \equiv \langle\langle width : unit \to int; height : unit \to int; area : unit \to int; \\
&\qquad\qquad\qquad equal : Rectangle \to bool\rangle\rangle
\end{aligned}$$

From a different perspective, we might be concerned as to which implementations produce objects of a certain type. In other words, given a specific object type, we are interested in whether some given implementation produces objects of this type. Basically, if all instances of an implementation I belong to an object type τ, then I *implements* τ [Ame90]. We can base the *implements* relationship on the subtype relation [AL90]:

Implements Relationship: An implementation I implements an object type τ if every instance of the implementation I belongs to the object type τ. According to the definition of the subtype relation (Section 2.2.2), this is exactly the case when the object type defined by I (meaning the inherent type of all of I's instances) is a subtype of τ[7].

The implements relationship is based on substitutability: an implementation I implements a type τ if I's instances can be used in any context that expects objects of type τ. An implementation may define more methods than required by the object type it implements, and the common methods may be in a subtype relationship—the function type of the method in I must conform to the signature of the message in τ [Por92]. This stems directly from the definition of subtyping for object types.

Example 3.5: Structural implements relationship.
Consider the following interface and implementation:

```
interface Dolly
      clone() returns Dolly
end Dolly
```

[6] We may confidently use the same name to denote both an implementation and the type it defines, since, due to our clear separation of implementations and types, possible ambiguities can always be resolved by the context in which the names appear.

[7] The relationship between an implementation and its type can also be described in terms other than the inherent type of the implementation's instances. For example, in models where an implementation does not automatically define an object type, the relationship between an implementation and its type may be established by a separate mechanism for individually combining types and implementations—each implementation defines projections from the type's signatures to its own methods [Hau93].

```
implementation MusicalSheep(pitch : int)
    var pitch : int := pitch
    method clone() returns MusicalSheep
        return new MusicalSheep(pitch)
    method bleat() returns nothing
        % bleat something ...
    private method think() returns nothing
        % empty!
end MusicalSheep
```

The private method *think* is not part of the public interface and, therefore, not a constituent of the object type. The interface corresponds to the recursive object type *Dolly* and the implementation is automatically associated with the recursive object type *MusicalSheep*:

$$Dolly \equiv \langle\langle clone : unit \rightarrow Dolly \rangle\rangle$$
$$MusicalSheep \equiv \langle\langle clone : unit \rightarrow MusicalSheep ; bleat : unit \rightarrow unit \rangle\rangle$$

According to the subtyping rule for recursive object types (Section 2.3), *MusicalSheep* is a subtype of *Dolly*. This confirms that the implementation *MusicalSheep* implements the interface *Dolly*, meaning that instances of *MusicalSheep* can be used in any context expecting objects of type *Dolly*.

A crucial consequence of defining the implements relationship in terms of the subtype relation is that any subsequent change or extension to the subtype relation for object types will have an immediate impact on the implements relationship [AL90]. In particular, [Ame90, LW90, LW94] formally describe the behavior of methods by specifying restrictions on the argument and result values. These behavioral constraints must be taken into consideration when verifying or deriving subtype relations. Therefore, assuming an implementation I is alleged to implement a type τ constrained by a semantic specification θ, then I must also satisfy the behavioral constraints described by θ.

In Section 2.2.5 we discussed the various outgrowths of implicitly inferred and explicitly declared subtype relationships. Following the same line of thought, and having based our definition of the implements relation on the subtype relation, it should be apparent that the issues pertaining to that discussion are equally applicable to the implements relationship. Therefore, depending on whether subtyping is structural or named, the adherence of a particular implementation to an object type may be structurally inferred by the compiler [AL90, BR97] or explicitly declared by the programmer [Por92, DGLM95, CCHO89].

In case of a hybrid type system, as suggested at the end of Section 2.2.5, the programmer may choose freely between inferred and explicitly declared implements relationships[8]. For example, the implements relationship could be inferred by default, but, for the convenience of potential readers, the implementation may explicitly declare which type it implements using an `implements` clause[9]. These declarations could be checked by the compiler. A similar approach is taken in [Ame90].

[8]Some approaches use conformance (structural subtyping) for type checking of assignments, but otherwise employ explicitly declared implements relationships [KMN+90].

[9]This would mean that an implementation I implements *only* the specified type τ directly, and other types through transitivity.

Clearly, since a type is automatically associated with each implementation, the implementation needn't explicitly state that it implements the "inherent" type of its instances. Further, declaring that a given implementation I implements a specific type τ implicitly declares that the type associated with I is a direct subtype of τ. Similar to named subtyping, an explicitly declared implements relationship between an implementation I and a type τ is legal only if the type associated with I conforms to τ.

Example 3.6: Named implements relationship.
The example with sheep given above assumed structural subtyping. In the following example the implements relation is explicitly stated[10]:

```
implementation Wolf implements MusicalSheep
    method clone() returns Wolf
        return new Wolf
    method bleat() returns nothing
        self.eatChalk()
        % try to bleat ...
    private method eatChalk() returns nothing
        % eat some chalk ...
end Wolf
```

The associated object type is:

$$Wolf \equiv \langle\langle clone : unit \to Wolf; bleat : unit \to unit\rangle\rangle$$

As pointed out above, the `implements` clause implicitly declares the type associated with the implementation *Wolf* to be a subtype of *MusicalSheep*. According to the subtyping rule for recursive object types (Section 2.3), the type *Wolf* conforms to *MusicalSheep*, meaning that the implicit subtype declaration is legal. Therefore, the implementation *Wolf* really does implement the type *MusicalSheep* and the `implements` clause is legal. Interestingly, the implementation *Wolf* also implements the type *Dolly*: since *Wolf* is a declared subtype of *MusicalSheep* and *MusicalSheep* conforms to *Dolly*, *Wolf* is a subtype of *Dolly* (due to transitivity).

It is widely accepted that separating types from implementations is beneficial [LSAS77, CCHO89, Por92]. Types and implementations describe different views of objects [SCB+86]:

1. A type specifies the behavior of its elements, meaning the externally observable behavior of the type's objects, without referring to implementational issues. A type represents the client's view of an object.

2. An implementation specifies the implementation of its instances, meaning the internal representation in terms of instance variables and algorithms in terms of method bodies. An implementation represents the implementor's view of an object.

Due to the precise distinction between types and implementations, the implements relationship between types and implementations is many-to-many: several implementations may implement a single object type and several object types may be implemented by a single implementation [BHJL86, AL90, PS92, Lam93]:

[10]Let us try to forget that the historical dispute actually saw goats as the wolf's adversaries.

- If a given implementation I implements a specific type σ, then, due to the definition of the implements and the subtype relation, the implementation I automatically implements all supertypes of σ [AL90], meaning all types τ with $\sigma \leqq \tau$. For example, this was illustrated by the implementation *Wolf* declared to implement the type *MusicalSheep*, but clearly also implementing all supertypes of *MusicalSheep*.

- More generally, with structural subtype and implements relationships, an implementation I implements all types to which the type associated with I conforms, meaning all types τ with $I \leqq \tau$. For example, in addition to *Dolly*, the implementation *MusicalSheep* also implements the type $\mathit{Sheep} \equiv \langle\langle \mathit{bleat} : \mathit{unit} \rightarrow \mathit{unit} \rangle\rangle$. Instances of a single implementation may belong to different and possibly unrelated (in terms of the subtype relationship) object types [PS92].

- In the case of explicitly-named implements relationships, an implementation may state its adherence to several different (maybe unrelated) types it implements. On the other hand, the implementation may be confined to name a single implemented object type. However, this type may actually have several supertypes, effectively making the implementation an implementation of all those supertypes [Por92].

- Several implementations may have completely distinct internal structures in terms of instance variables and method implementations, but nevertheless all implement the same interface [BCC+96].

Example 3.7: Implementations of type *Point*.
The type *Point* (Example 2.18) can be implemented in different ways:

```
implementation PolarPoint(x : real, y : real) implements Point
    var rho : real := sqrt((x*x) + (y*y))
    var theta : real := argtangent(y/x)
    method x() returns real
        return rho * cosine(theta)
    method y() returns real
        return rho * sine(theta)
    method equal(p : Point) returns bool
        return self.x() = p.x() and self.y() = p.y()
end PolarPoint

implementation CartPoint(x : real, y : real) implements Point
    var x : real := x
    var y : real := y
    method x() returns real
        return x
    method y() returns real
        return y
    method equal(p : Point) returns bool
        return self.x() = p.x() and self.y() = p.y()
end CartPoint
```

As opposed to traditional systems based on abstract data types [LSAS77], in our system multiple implementations of the same type may simultaneously coexist and are dynamically interchangeable for one another [KMN+90, BR97, DGLM95]. Both implementations are clearly distinct but nevertheless implement the same type *Point*. A user of an object belonging to type *Point* should not be concerned with whether the object is actually an instance of *PolarPoint* or *CartPoint*. Although their objects have different instance variables, the code of *equal* is identical for both implementations. In Chapter 4 we will try to exploit such similarities by promoting the reuse of existing code and reducing the overall maintenance effort.

The implements relationship is not really symmetrical [AL90]. A given implementation can be associated with a unique *most specific* type, namely what we have repeatedly called the "inherent" type of the implementation's instances. However, the opposite direction is not quite so unambiguous: as demonstrated by the prior example, a given type does not automatically imply a specific implementation. This is perfectly obvious, since our main premise was to decouple types from implementations. When dealing with objects of a given type we are, in general, not interested in how these objects are individually implemented. In particular, although all instances of a implementation I irrefutably belong to I's associated type, not all elements of this type must necessarily be instances of the implementation I, because other implementations may also happen to implement the same type.

Chapter 4

Inheritance

One of the most conspicuous concepts in object-oriented programming is the notion of *inheritance*, first introduced in the 1960s as a major feature of the programming language Simula[1] [DN66]. In fact, inheritance is commonly regarded as a (if not *the*) central feature to object-orientation [Car88, CP89, JA92, DT92, Sin95], often being acclaimed as the solution to many of the problems hampering software development [Tai96]. Inheritance constitutes a powerful mechanism for structuring large programs and its presence has decisively determined the success of object-oriented programming [Ame87].

Cardelli [Car88] suggests that an examination of different object-oriented languages shows that inheritance is the *only* notion critically associated with object-oriented programming. This view is backed by the ubiquitous presence of inheritance in many introductory texts on object-orientation [Str87, Eck95, Mey97], crucially distinguishing object-oriented programming from other well-established or modern programming paradigms [Win92].

However, it has been argued that inheritance is *not* the only notion intrinsic to object-oriented programming [Ame87, Rum93]. The quintessence of object-orientation lies not in the stringent presence of inheritance, but more in the consistent interplay between a subset of features such as inheritance (in the sense of hierarchical code sharing [Cook92, Por92]), data abstraction [LZ74, LSAS77], the separation of interfaces from implementations [Lis87, CCHO89], encapsulation of data and operations in objects and protection of individual objects against each other (originating from abstract data types) [Sny86, Ame87], subtype polymorphism [Car88, BCC+96], parametric polymorphism [CW85, DGLM95], and object identity [Rum93]. Interestingly, many of the existing widely acknowledged object-oriented languages lack one or more of these features. For example, Java [GJS96] provides no support for parametric polymorphism[2], Emerald [BHJL86] lacks inheritance, and Simula [DN66] and Flavors [Moon86] elide encapsulation. All the same, these programming languages are (with a few exceptions [Weg88]) commonly accepted as being object-oriented.

As documented by [Cook87a, Weg88, PS92], inheritance is a controversial and inadequately understood mechanism—researchers rarely agree on its meaning [Tai96], let alone on the convention of terminology in use. Despite the fact that much effort has been invested into research on inheritance, there is no universal consensus on the definitions or mechanisms

[1] At that time, inheritance was also known as "concatenation" or "prefixing", since it was informally defined in terms of textual concatenation of program blocks [CP89].

[2] Recently, several proposals have been issued to add parametric polymorphism to the Java programming language [AFM97].

which should be used to support it [JA92], evident from the proliferation of formulations of inheritance that have been proposed [CP89].

Nevertheless, when the phrase "object-oriented" is mentioned, probably the first feature most people will think of is inheritance [Rum93], many of the alleged benefits of the object-oriented paradigm (such as enhanced reusability and maintainability) being largely accredited to it [Tai96]. Inheritance is an important concept and a useful mechanism for structuring large systems [Ame87]. Therefore, this section will concentrate on the notion of inheritance. In particular, we will generalize the common view of inheritance as a mechanism for reusing existing implementation code (Section 4.3) by extending it to encompass the reuse of existing type specifications (Section 4.4). The latter will be clearly distinguished from subtyping, hopefully alleviating a frequent misunderstanding of the two concepts. To start with, we will introduce an abstract form of inheritance independent of particular entities, and then illustrate the inheritance mechanism by concretely applying the previously gained results to produce implementation inheritance and type inheritance. For a detailed and comprehensive study of inheritance the reader is referred to [Tai96].

4.1 Incremental Modification

The basic idea of inheritance seems fairly simple, allowing new entity definitions to be based on existing definitions [SCC+93]. When a new entity is to be defined, only the characteristics distinguishing it from those of some existing entity specifications need to be declared explicitly. Other properties are automatically *inherited* from and shared with the existing entities [Tai96].

Inheritance is a special relationship between entities, in which one entity receives properties or characteristics from another entity [Tai96]. The receiver's properties may diverge from the giver's properties in some particular way, meaning that the receiver may have characteristics additional to those inherited from the giver, or the receiver may replace some inherited properties with alternative versions. Moreover, not all properties must necessarily be inherited by the receiver [DT88, LT93]. Inheritance may be *single* or *multiple*, depending on whether an entity is restricted to inherit from only one single parent entity, or permitted to inherit from multiple parent entities [Car88]. Multiple inheritance is treated in [BI82, Sny86, CG90, Sin95, RFW96, Tai96, Mey97, Str97].

In the context of programming languages, we regard inheritance as a mechanism for *differential* or *incremental* programming [CP89]. New program components are constructed by specifying how they differ from existing program components [BPS89, JA92, SCC+93]. In particular, inheritance allows the definition of new software elements as *modifications*, *extensions*, or *restrictions* of previously defined ones, which should not have to be modified for the occasion [Mey88]. In this sense, inheritance is a technique for *code reuse* or *code sharing* [LP91, Tai96], aiming at the reduction of the overall implementational effort and the improvement of a program's maintainability and flexibility[3] [Sin95, CRR98]:

- New program units are defined in terms of old ones. This sharing of program code can result in a considerable reduction of the overall program size.

[3]The benefits of inheritance are greatly reduced in concurrent object-oriented programming, due to a phenomenon usually termed *inheritance anomaly* [Frø92]. See [CRR98] for a formal analysis.

- Changes to one program unit are automatically propagated to all program units inheriting from it[4]. Therefore, extensions and modifications to a set of entities can be localized to a single specification.

The ease with which an existing software system can be extended and modified is symptomatic for how effectively it will accommodate the evolving requirements in the future. Traditional programming methodologies do not harmonize well with this evolutionary nature of software, typically relying on implementation techniques such as tag-analyses or "copy-and-paste" techniques [Str87, Tai96].

Before turning to concrete forms of inheritance, such as implementation inheritance (Section 4.3) and type inheritance (Section 4.4) which operate on particular program units (implementations and types, respectively), we will define the semantics of inheritance as a general mechanism which facilitates the sharing of properties between abstract entities:

Inheritance: Inheritance combines a parent entity P and a modifier M to produce a new result entity R, formally $R = P \oplus M$ [WZ88, CP89]. The result entity R *shares* the properties of P, taking into account the modifications specified by M. The new entity R is defined only by specifying the concrete differences M from P [Sei87].

To simplify the presentation, we will assume that the entities involved and being manipulated can be represented as record-like structures with finite numbers of attributes[5]. For example, the entity P can be represented as $P = \{p_1, \ldots, p_k\}$, whereby the attributes $p_1, \ldots, p_k$ correspond to the properties of the entity P.

The combination operator $\oplus$ is asymmetrical, since the parent P and the modifier M play different roles in the composition process. It describes how the attributes of P are to be modified by M. Typically, M will incrementally add new properties or will introduce properties that overlap with those of P [Tai96]:

1. If the attributes $p_1, \ldots, p_k$ of P and $m_1, \ldots, m_l$ of M are disjoint ($p_i \neq m_j$ for all $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, l\}$), then R has $k + l$ attributes consisting of the union of the attributes of P and M. The entity R *inherits* (shares) the attributes of P, augmented by the *locally* defined attributes specified by M. The resulting entity is $R = \{p_1, \ldots, p_k, m_1, \ldots, m_l\}$.

2. If the attributes of P and M overlap ($p_i = m_j$ for some $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, l\}$), then R has the attributes $m_1, \ldots, m_l$ of M and those attributes p_i of P that do not occur in M ($p_i \notin \{m_1, \ldots, m_l\}$). The attributes of the modifier *override* (i.e. take precedence over) the equivalent attributes of the parent entity.

Example 4.1: Abstract inheritance.
Consider the following entities *Parent* and *Modifier* with attributes representing some arbitrary characteristics:

[4] Apart from the reduction of code, the clear advantage of sharing code over copying this code is that modifications to the code can be performed by modular modification of a single program unit, rather than modifying each copy separately [WZ88, CP89, Tai96].

[5] The term "attribute" should be understood fully abstractly, denoting any possible kind of entity characteristic. When applying inheritance to implementations (or types), attributes will correspond to methods and variables (or signatures).

$Parent = \{p_1, p_2\}$
$Modifier = \{p_2, m_1\}$

A new entity *Result* can be constructed by inheriting and sharing the properties of *Parent*, modifying them as specified in *Modifier*:

$Result = \{p_1, p_2, m_1\}$

The attribute p_1 is inherited from the entity *Parent*. The entities *Modifier* and *Parent* overlap in the attribute p_2, meaning that as part of the inheritance process $Parent \oplus Modifier$ the attribute p_2 is *overridden* by *Modifier*. Therefore, *Result* has the attribute p_2 occurring in *Modifier* and not in *Parent*. Finally, *Modifier* introduces a new property, denoted by the attribute m_1.

Inheritance Hierarchy: Entities obtained through inheritance, such as $R = P \oplus M$, can themselves be used as parent entities: $R_1 = R \oplus M_1$. This reiterative application of the inheritance operator leads to an *inheritance hierarchy* which forms a tree [SCC+93], or, more generally in the case of multiple inheritance, a directed acyclic graph[6] [Car88]. Every node in the graph is an entity that represents some collection of properties defined incrementally in terms of the entity's parents [Tai96].

Clearly, modification is simpler for independent attributes, exhibiting maximum code reuse since the parent's properties are completely reused [CRR98]. Nevertheless, software evolution, in general, requires the modification of existing attributes rather than merely adding new ones [WZ88].

4.2 The Essence of Inheritance

The entities involved in inheritance may, in general, have recursive structures [WZ88, CHC90, CCHO89]:

Self-Referential Entity: An entity P is *recursive* or *self-referential* if some attribute p_i of the entity P refers to P itself. This recursion may generally be denoted by the occurrence of *self* in p_i.

Example 4.2: Recursive entity.
Consider the following entity:

$P = \{p_1 : \ldots, p_2 : \ldots\, self.p_1 \ldots\}$

The entity P is self-referential in the attribute p_2, accessing its own attribute p_1 by recursively referring to itself via *self*.

Recursive entities are very common in object-oriented programming [Tai96]. In fact, we have already encountered recursive types (Section 2.3) and recursive objects (Chapter 3):

[6]The graph is required to be acyclic, since it is nonsensical for an entity to inherit properties from itself [Tai96].

1. The interface *Bag* in Example 2.15 is recursive in the attributes corresponding to the signatures *put*, *equal*, and *union* (denoted by the occurrence of *Bag*).

2. The implementation *Rectangle* in Example 3.2 creates objects which are recursive[7] in the attributes corresponding to their methods *area* and *equal* (denoted by the occurrence of `self`). In addition, the type *Rectangle* associated with the implementation is also recursive in the attribute corresponding to the signature *equal* (denoted by the occurrence of *Rectangle*).

As a prelude to elaborating the inheritance mechanism for recursive structures, and to illustrate the intricacies encountered therein, we will motivate the reader with an example dealing with recursive functions (similar to [CP89]):

Example 4.3: Deriving recursive functions.
Consider the function f which, in the course of computation, recursively invokes itself:

```
function f ()
    ...
    f()
    ...
```

Assume the necessity to define some other function g, which, in principle, performs the same computations as f, but precedes them by some special computations. It would be convenient to reuse the implementation of f, meaning that g performs the special computations and then invokes f—a user of g needn't know whether g is implemented in terms of f or from scratch.

```
function g ()
    ... % additional computations
    f()
```

However, this naïve solution has introduced a serious problem: the additional computations of the function g are only executed once, namely in the first invocation of g—the recursive function calls in f still invoke the original function f. In general, the function g will expect the special computations to be performed in each recursive invocation. The solution using f is not equivalent to a solution from scratch in which g would invoke itself. Therefore, a mechanism is required to explicitly capture self-references and to reinterpret them under inheritance [WZ88, CP89].

The prior example demonstrated that the fundamental and essential aspect of inheritance is concerned with the interplay between inheritance and self-references [WZ88, CCHO89, CP89, Tai96]. Assume that an entity P is self-referential in some attribute p_i. When a new entity R is constructed by inheriting from (and sharing the attributes of) P, possibly modified by some modifier M, which entity should the self-reference in p_i denote when p_i is invoked on account of R?

Example 4.4: Inheritance and recursion
For example, assume a parent entity similar to Example 4.2 and a modifier entity:

[7] Note that, while the instances of *Rectangle* have recursive structures, the implementation itself is *not* recursive, meaning the implementation *Rectangle* does not refer to itself as an implementation. This type of recursion is dealt with in [CCHO89, CHC90, PS91].

$$Parent = \{p_1 : \ldots, p_2 : \ldots self.p_1 \ldots\}$$
$$Modifier = \{p_1\}$$

The attributes of *Parent* can be modified by *Modifier* to produce a new entity $Result = Parent \oplus Modifier$. The attribute p_2 is inherited from and shared with the entity *Parent*. The attribute p_1 is overridden by *Modifier*, meaning that *Result* has the attribute p_1 specified by *Modifier*:

$$Result = \{p_1, p_2\}$$

If the attribute p_2 is accessed on behalf of *Parent*, then the self-reference in p_2 should naturally denote *Parent*, and in further consequence access the attribute p_1 defined in *Parent*. However, if the attribute p_2 is accessed on behalf of *Result*, then the self-reference in p_2 should now denote *Result* (although p_2 is actually defined in *Parent*), so as to refer to the *overridden* attribute p_1 defined in *Modifier*.

Thus, the entity actually denoted by the self-reference changes under inheritance [CP89]. When the attribute of some entity R (obtained by inheritance: $R = P \oplus M$) is accessed, the self-references in that attribute must be bound to R, regardless of whether the attribute is actually defined in P or M. A self-reference in some attribute of P, such as $self.p_1$ in Example 4.4, is bound to the definition of p_1 in P, or, if p_1 is overridden by M, to the definition in M.

The attributes defined in the parent entity P are *internalized* by all entities R inheriting from P [Weg87]. These attributes can be used by R as if they were defined locally in R and not inherited from P. The self-references in these inherited attributes refer to the entity on whose behalf they were accessed rather than referring to the textual module in which they were originally defined [CP89]. Therefore, the attributes can be shared by inheriting entities and behave like indigenous attributes of each inheritor [WZ88].

Inheritance is a mechanism for deriving modified versions of recursive entities [CP89] and is intimately connected to self-references. The self-references in the inherited attributes must be changed to refer to the modified (overridden) definitions [CP89, CCHO89, Tai96]. By virtue of inheritance, a new entity can be defined by internalizing properties of other entities as though they were its own [WZ88].

It is important to point out that the binding of a self-reference to a particular entity may be performed at run-time *or* at compile-time, depending on the concrete entities being manipulated. For example, objects are typically dynamic entities; when object implementations are inherited, the self-references in the inherited methods can only be bound to the correct object at run-time. On the other hand, types are static entities; when type specifications are inherited, the self-references in the type's constituents can be bound to the correct types at compile-time.

The employment of inheritance as a code reuse mechanism should always keep a balanced equilibrium between the inevitably entailed increase in program complexity and the potential benefits of reduced program size. The use of inheritance may have both positive and negative impacts on the maintainability and structure of a program:

+ The overall program size is reduced.

+ New entities can be explained in terms of already existing and well-known entities, making them easier and faster to understand.

+ Changes to a set of related entities may be performed by locally changing a single parent entity.

– Inheritance relationships must be kept track of by the implementor, requiring an increased administrative effort (especially in the presence of independent inheritance and subtype hierarchies).

– The implementor must invest special care in the manipulation of program entities, since every entity has potential child entities which would be affected by the changes.

The following sections will map the inheritance mechanism defined for abstract entities onto more concrete program entities, namely implementation specifications and type declarations.

4.3 Implementation Inheritance

Where does it make sense to apply inheritance in object-oriented programs, or, in other words, which concrete program units can reasonably be shared? We will start by exploring the traditional reuse of object implementations, and later in Section 4.4 continue by generalizing our understanding of inheritance to encompass the inheritance of type specifications.

An individual method or variable declaration does not seem to constitute a reusable module independently of the implementation to which it belongs. Therefore, the unit of reuse considered in this work is the entire implementation specification, providing a relatively appealing level of granularity[8] [Mey88]. The sharing of implementation specifications can dramatically reduce the amount of code in a system [Ame90, LP91].

The abstract inheritance mechanism introduced in Section 4.1 can straightforwardly be rendered to the concrete inheritance of object implementations: the entities being manipulated by inheritance are implementation specifications; the attributes being inherited and shared are the instance variable declarations and method definitions provided by these implementation specifications [HB87, Por92]. Implementation inheritance composes new implementations using already existing ones [Ame87, Hau93, Tai96] by extending or refining the inherited representation and algorithms [Horn87].

In conventional object-oriented jargon this form of inheritance is also termed *subclassing*[9] [Ghe91, LP91, PS92, BCC+96]. Following this nomenclature, the entity whose properties are inherited is termed *superclass* and the entities inheriting these properties are termed *subclasses*. Having replaced the term “class” with “implementation”, we refer to the corresponding entities as *subimplementations* and *superimplementations* [Por92], directly corresponding to the result and parent entities from Section 4.1.

Implementation inheritance is an incremental modification technique for reusing existing object descriptions, expressing new implementations in terms of old ones [CCHO89, CP89, Cus91]. A new implementation I can be defined by inheritance in terms of an already existing implementation I' by specifying only the incremental differences between I and I' [Ghe91]. The subimplementation I inherits all instance variable declarations and method definitions

[8] Other studies, for example [Ame87], propose *inheritance packages* which are completely independent of implementations.

[9] Smalltalk-80 [GR83] uses the term “subclassing”, while C++ [Str97] and Eiffel [Mey97] use “derivation” and “inheritance”, respectively [SCC+93].

from its superimplementation I' [SCB+86], and additionally specifies which new instance variables and methods should be added to the instances of the subimplementation I in comparison to the instances of the superimplementation I' [AL90, Ghe91, LP91, Por92, BCC+96]. Therefore, an instance of a subimplementation I typically has both the variables and methods defined in the superimplementation I' as well as the variables and methods defined locally by the incremental modifier associated with I [DT88].

As specified in Section 4.1, the attributes of the superimplementation and the modifier associated with the subimplementation may overlap: the attributes of the modifier are *redefinitions* of (and override) the equivalent attributes of the superimplementation[10]. In this sense, a subimplementation may provide reimplementations of the methods inherited from the superimplementation [AGS91, Ghe91]: the subimplementation overrides the definitions of the methods it would otherwise inherit by redefining them[11] [SCB+86, BP93, BCC+96].

A subimplementation can have its own instance variable and method definitions (which may themselves be inherited from other implementations or defined locally) inherited by further subimplementations [HB87, Weg87, BGM89]. The hierarchy resulting from this repeated application of implementation inheritance is commonly known as an implementation hierarchy [Ame90, Por92]. Implementation hierarchies are incrementally and explicitly defined using the inheritance mechanism [BGM89], which allows one object implementation to be based on another hierarchically [Lis87]. An important goal during implementation is to identify an implementation hierarchy that simplifies program maintenance and future modifications [Lis87].

Inheritance is a key structuring mechanism in object-oriented programming languages [Frø92], supporting reusability and convenience for the implementors [LP91, Tai96]. The programmer inherits from some existing implementation that is close to the desired target implementation and modifies or adds new attributes [Rum93]. Thus, the overall program size can be reduced considerably, resulting in programs which are easier to read and maintain.

Example 4.5: Implementation inheritance.
The implementation *Square* describes objects representing squares:

```
implementation Square
    var side : int
    method side() returns int
        return side
    method area() returns int
        return side * side
    method equal(other : Square) returns bool
        return side = other.side()
end Square
```

The implementation *ColorSquare* inherits the instance variable and the method definitions from *Square*, adds an additional instance variable to denote the square's color, and overrides the definition of *equal* to consider this color during comparison:

[10] Since we consider inheritance merely as a mechanism for code sharing, we impose no a priori restrictions on the redefinitions of methods in a subimplementation. Nevertheless, in most object-oriented programming languages, inheritance is coupled to subtyping, leading to stringent constraints when redefining methods.

[11] A method definition m_P in the superimplementation is overridden by a method definition m_C in the modifier if m_P and m_C have the same signatures.

```
implementation ColorSquare inherits Square
    var color : int
    method color() returns int
        return color
    method equal(other : ColorSquare) returns bool
        return side = other.side() and color = other.color()
end ColorSquare
```

These implementations correspond directly to the entities used in the definition of inheritance in Section 4.1: *Square* is the parent entity, *ColorSquare* is the result entity, and the variable and method definitions following the `inherits` clause constitute the modifier entity. This modifier describes *ColorSquare* as a new collection of properties in terms of the incremental differences to *Square*.

As elaborated in Section 4.2, it is indispensable for the inherited attributes to behave as though they were defined in the inheritors themselves. Related to the inheritance of implementation specifications, this signifies that subimplementations must internalize inherited methods as though they were their own—the inherited methods must behave like indigenous methods of each subimplementation.

The abstract entities encountered earlier in the definition of inheritance had self-referential attributes. Similarly, an object may have a recursive structure, meaning that the object refers to itself in one of its method bodies. This recursion can be made explicit by the occurrence of the pseudo-variable[12] `self` in the method body provided by the implementation specification [BCC+96]. The pseudo-variable `self`[13] denotes the receiver of the original message [CCHO89]—the object on whose behalf the current method is being executed.

Implementations describe dynamic entities, namely objects which are created at run-time and share the methods specified by the implementation specification. Therefore, the pseudo-variable `self` in some method definition cannot be bound to a concrete object at compile-time, but is bound to the message receiver when the method is invoked at run-time (*late binding*).

Example 4.6: Inheritance of recursive object implementations.
Assume that some implementation I defines a self-referential method m, and that this method is inherited by some subimplementation of I, say I'. If an instance of I' receives the message associated with m, then `self` in m will be bound to the instance of the subimplementation I', even though m was originally defined for instances of the superimplementation I. The self-reference changes under inheritance, denoting instances of different implementations depending on the message receiver. In essence, implementation inheritance is a mechanism for deriving one implementation specification from another, in which self-references are late-bound [BP93, Tai96].

To illustrate the inheritance of recursive object implementations more concretely, we will employ a frequently stressed example concerned with graphical windows [Cook87a, Tai96]:

[12] See Section 3.1 for a brief description of pseudo-variables.

[13] The pseudo-variable `self` is used by Smalltalk-80 [GR83] and derivatives. C++ [Str97] and Java [GJS96] use `this` and Eiffel [Mey97] uses `current`. In C++, Java, and Eiffel the self-references are usually implicit: late-bound self-referential messages look syntactically like standard procedure calls by eliding the explicit occurrence of `this` or `current`, respectively.

Example 4.7: The implementations *TextWindow* and *TitledTextWindow*.
The following implementation *TextWindow* describes objects representing graphical windows containing text:

```
implementation TextWindow
    var position : Point
    var frame : Rectangle
    var contents : string
    method drawFrame() returns nothing
        % draw the frame of the window at the window's position
    method drawContents() returns nothing
        % output the contents inside the window
    method refresh() returns nothing
        self.drawFrame()
        self.drawContents()
end TextWindow
```

Instances of *TextWindow* are recursive: the self-references in *refresh* denote the message receiver—the object on whose behalf *refresh* is being executed. At run-time, when an instance of *TextWindow* receives the message *refresh*, `self` will be bound to that object. The object sends itself the messages *drawFrame* and *drawContents*.

The subimplementation *TitledTextWindow* inherits all instance variables and method definitions from its superimplementation *TextWindow* and overrides the definition of *drawFrame* to include the title in the frame:

```
implementation TitledTextWindow inherits TextWindow
    var title : string
    method drawFrame() returns nothing
        % draw the frame and the title of the window
end TitledTextWindow
```

When an instance of *TitledTextWindow* receives the message *refresh*, it will invoke the corresponding method inherited from and shared with *TextWindow*. However, the self-reference in the method will now be bound to the instance of *TitledTextWindow*, even though *refresh* was originally defined for an instance of *TextWindow*. Therefore, the message *drawFrame* will be sent to an instance of *TitledTextWindow*, invoking the redefinition of *drawFrame* in *TitledTextWindow*. By virtue of late-binding, *TitledTextWindow* can internalize the method *refresh* as though it were not inherited from *TextWindow* but defined locally.

The language mechanism of late-binding allows the resolution of references to properties of objects to be postponed until run-time[14]. In particular, recursive self-references to other properties of the same object are not statically fixed but may invoke different methods depending on the dynamic context from which the message is sent. This dynamic context is specified by the pseudo-variable `self`, which may denote instances of different implementations at run-time. Implementation inheritance with late-binding minimizes the need for physical code duplication by allowing a single method to invoke different methods depending on the dynamic context denoted by `self` [Weg87, Tai96].

[14] Note that implementation inheritance and late-binding are independent of whether a language is strongly-typed or dynamically-typed, being equally present in both [Ame90].

Some researchers argue against the use of inheritance of implementation code, suggesting that the same effect could be achieved with composition and forwarding [Lis87, DT92]:

Example 4.8:
The prior implementation could be specified without inheritance:

```
implementation TitledTextWindow
    var title : string
    var textWindow : TextWindow
    method drawFrame() returns nothing
        % draw the frame and the title of the window
    method drawContents() returns nothing
        textWindow.drawContents()
    method refresh() returns nothing
        textWindow.refresh()
end TitledTextWindow
```

This (admittedly) naïve solution using composition does not work, because the self-references are not treated correctly [Cook87a]: when an instance of *TitledTextWindow* forwards the *refresh* message to the *TextWindow* subobject, `self` will denote the subobject instead of the *TitledTextWindow* instance, and therefore invoke the "wrong" *drawFrame* method. A proper solution, simulating the *delegation* mechanism in prototype-based languages [Lie86, US87, Weg87], is illustrated in [JZ91, Dav92].

Frequently, inheritance is motivated by referring to the classification of concepts in taxonomic systems [Win92], attempting to model and describe real world application domains in terms of implementation hierarchies [Weg87]. We argue that application domains should best be described by type hierarchies and therefore be independent of implementation hierarchies (see Chapter 5 for a detailed discussion). Implementation inheritance is purely a low-level mechanism for reusing existing implementation code [AL90, Tai96], and as such serves to classify implementations by their shared specifications (expressed by the implementation hierarchies). There is a growing consensus that implementation inheritance (and inheritance in general) is a *producer's mechanism*, being fairly independent of a client's view or use of objects[15] [Cook92]. In particular, inheritance does not provide a classification of the externally observable behavior of the implementation instances.

The clear distinction between types and implementations suggests independent implementation and subtyping hierarchies [Har91]. Nevertheless, many object-oriented programming languages such as C++ [Str97], Eiffel [Mey97], and Trellis/Owl [SCB+86] identify the concepts of subtyping and inheritance [CHC90]. Section 4.4 and Chapter 5 will explore whether this approach is justified:

1. Does a subimplementation always produce a subtype? If the implementation I has a subimplementation I', then is I''s inherent type always a subtype of I's inherent type?

2. Does a subtype always imply a subclass? If type τ is implemented by an implementation I, then is τ's subtype σ always implemented by a subimplementation I' of I?

[15] Typically, the only place where a client actually needs to see a concrete implementation name is on instance creation [Sou97]. From that point onwards, the specific implementation of the object "vanishes" behind the coat of protection provided by types.

We shall see that implementation inheritance and subtyping are fully independent notions, leading to completely decoupled implementation and subtype hierarchies. In particular, we will conclude that while an implementation does not necessarily implement a subtype, it implements an inherited type [CCHO89, CHC90]: implementation inheritance is related to *type inheritance* [Hau93]—a notion dealt with next.

4.4 Type Inheritance

In sections 4.1 and 4.2, inheritance was introduced as a general mechanism for reusing (possibly recursive) program components. The previous section applied this abstract inheritance mechanism to the reuse of object implementations, describing the form of inheritance most frequently encountered in practice. However, the inheritance mechanism can also be meaningfully applied to type specifications. In this sense, we part from the common view of inheritance as a facility confined merely to implementation specifications, and apply the abstract inheritance mechanism to the reuse of interfaces.

Type inheritance is particularly appealing in domains primarily concerned with the specification of interfaces and their relationships, such as interface definition languages [KLMM94]. This study describes a similar environment in which the specification of interfaces and the definition of implementations are fairly independent notions. To illustrate the necessity and the preferences of type inheritance in such an environment, consider the following example:

Example 4.9: The interfaces *2DPoint* and *3DPoint*.
The interfaces *2DPoint* and *3DPoint* describe objects representing 2-dimensional and 3-dimensional points, respectively:

```
interface 2DPoint
    x() returns real
    y() returns real
    equal(2DPoint) returns bool
end 2DPoint

interface 3DPoint
    x() returns real
    y() returns real
    z() returns real
    equal(3DPoint) returns bool
end 3DPoint
```

Apart from the additional message z, the interfaces *2DPoint* and *3DPoint* bear evident similarities:

- Both *2DPoint* and *3DPoint* have the messages x and y with completely identical signatures.
- The message *equal* is present in both interfaces, albeit in *2DPoint* it recursively refers to *2DPoint*, while in *3DPoint* it recursively refers to *3DPoint*. More generally, in both cases the signature of *equal* recursively refers to the interface being defined.

Keeping in mind the aim for reducing the overall implementational effort and increasing the maintainability and flexibility of programs, it would prove convenient to exploit these resemblances by expressing the interface *3DPoint* by describing the incremental differences to the interface *2DPoint*.

Type inheritance provides these facilities. In terms of the abstract inheritance mechanism, the entities being manipulated by type inheritance are interface specifications; the attributes being inherited and shared are the signatures. Typically, the inheritor may modify the inherited entity by adding new signatures or by overriding the inherited signatures. Overriding a signature involves changing the parameter or result types of the signature[16] [CCHO89]. Some environments even make possible the selective inclusion or exclusion of signatures [KLMM94].

The concept of type inheritance allows the reuse of type declarations, facilitating the composition of new interfaces using already existing ones [CCHO89, CHC90, Hau93]—one type definition is based on another. The underlying idea of type derivation is to allow a type declaration made in one place for some particular purpose to be reused in another place for a similar, related, or perhaps completely different purpose [KLMM94].

Similarly to implementation inheritance, type inheritance is a producer's mechanism. Clients should only be concerned with the subtype relationships between the defined types. As will be illustrated later, the subtype hierarchy is quite distinct and independent from the hierarchy induced by type inheritance [CCHO89, CHC90, Har91, KLMM94]. Inheritance is a technique for deriving one type from another—only in some cases will the derived specification actually be a subtype of the initial specification [ASB94]. Inheritance between types can be the basis for subtyping but the mere fact that one type is derived from another does not necessarily indicate that a subtype relationship exists between them [CHC90, Hau93, KLMM94]. In general, an attempt to use an object of a derived type where an object of a subtype is expected is a violation of substitutability and will be caught at compile-time [Har91].

Non-recursive signatures such as x and y in *2DPoint* may be inherited straightforwardly, the effect being basically equivalent to copying the declarations from one interface into another [KLMM94]. However, recursive signatures such as *equal* in *2DPoint* require greater attention. Taking into consideration the conclusions from Section 4.2 and Example 4.9, self-references in recursive interfaces should be reinterpreted under inheritance to denote the interface being defined [CCHO89].

Reinterpreting *every* self-reference under inheritance may be undesirable, since some interfaces may "incidentally" refer to themselves—such "self-references" may not be intended to denote the inheritor in the derived interface but should still refer to the parent interface. Instead, we seek a more general mechanism for explicitly denoting those self-references which should be reinterpreted under inheritance.

In the context of implementation inheritance, recursions in objects were captured by the pseudo-variable `self` in the method bodies. Similarly, in recursive interfaces, the self-references can explicitly be captured by `MyType` [AL90, BCC+96]. The pseudo-type `MyType` denotes the same type as the interface being defined [Har91].

Example 4.10: The interface *2DPoint*.
The recursive interface *2DPoint* from Example 4.9 can be respecified by making the self-reference explicit:

[16] Recall that overriding a method in implementation inheritance involves changing the method's implementation rather than its signature.

```
interface 2DPoint
      x() returns real
      y() returns real
      equal(MyType) returns bool
end 2DPoint
```

Example 4.11: *Bag* and *Set*.
The object types *Bag* and *Set* are almost identical except for the self-reference in *equal*:

$$Bag \equiv \langle\langle put : int \rightarrow unit; get : unit \rightarrow int; equal : Bag \rightarrow bool \rangle\rangle$$
$$Set \equiv \langle\langle put : int \rightarrow unit; get : unit \rightarrow int; equal : Set \rightarrow bool \rangle\rangle$$

Bag and *Set* could be expressed with a single object type definition by making use of `MyType`:

$$Bag \equiv \langle\langle put : int \rightarrow unit; get : unit \rightarrow int; equal : MyType \rightarrow bool \rangle\rangle \equiv Set$$

Interfaces that are self-referential via `MyType` work well with inheritance [BCC+96], because the self-references are reinterpreted under inheritance to refer to the inheritor [AL90]—the meaning of `MyType` changes to reflect inheritance [Har91]. A recursive signature can be shared between several interfaces and behave as an indigenous signature of each interface. `MyType` in the signature always denotes the interface being defined and not the interface in which the signature was textually declared. Type inheritance characterizes a collection of types with similar recursive structures [CCHO89].

Example 4.12: The interface *3DPoint*.
The interface *3DPoint* from Example 4.9 can be respecified by explicitly inheriting from the revised definition of *2DPoint*:

```
interface 3DPoint inherits 2DPoint
      z() returns real
end 3DPoint
```

The specification of *3DPoint* is minimized to the definition of the additional message z, achieving an optimal reuse of the type specification of *2DPoint*. When used in the context of *3DPoint*, the self-reference `MyType` in *2DPoint*'s *equal* message is changed to denote the inheriting interface *3DPoint*.

By virtue of the pseudo-type `MyType`, declaring an interface to inherit from another interface essentially has the same effect as inserting (copying) all signatures of the latter interface into the former interface [KLMM94]. Without the aid of `MyType`, recursive references would not denote the inheritor when inherited. `MyType` guarantees that self-references are treated appropriately. Type inheritance can be degraded to a purely syntactic mechanism—type derivation is equivalent to defining the inheritor in "flat" form without inheritance [Mey90], explicitly specifying all signature constituents [KLMM94].

The pseudo-type `MyType` is not confined to be employed only in interface specifications. Sections 3.1 and 4.3 demonstrate how the pseudo-variable `self` can be used in an implementation's methods to denote the receiver of the current message, meaning the object on whose behalf the current method is being executed. Likewise, `MyType` can equally be utilized in implementations to denote the *type* of the receiver of the current message—`MyType` in an implementation essentially denotes the type of `self` [BCC+96].

In principle, the pseudo-type `MyType` in an implementation specification refers to the inherent type of the implementation being defined—the type implicitly associated with the implementation's instances (Section 3.2) [AL90]. `MyType` can be deployed in the definition of methods whose parameter or return types are intended to be the same as the type of the message receiver [BCC+96].

The utilization of `MyType` in implementations interacts well with implementation inheritance: when an implementation specification defined in terms of `MyType` is inherited by a subimplementation, the meaning of `MyType` changes to denote the inherent type associated with the subimplementation [AL90, BCC+96]. Due to the fact that implementations define types, our form of implementation inheritance is related to type inheritance: the inherent type associated with a subimplementation can be obtained by inheriting from the inherent type associated with the superimplementation—subimplementations have inherited types [CCHO89, CHC90]. However, due to the covariant behavior of `MyType` on inheritance, the mere fact that one implementation inherits from another doesn't necessarily imply a subtype relationship between the associated object types [AL90].

Example 4.13: Implementation inheritance and type inheritance.
Consider the following interfaces and their implementations:

```
interface ParentType
    plok(MyType) returns MyType
end ParentType

interface ChildType inherits ParentType
    plok2() returns nothing
end ChildType

implementation Parent implements ParentType
    method plok(MyType) returns MyType
end Parent

implementation Child inherits Parent implements ChildType
    method plok2() returns nothing
end Child
```

The type implemented by *Child* can be obtained by type inheritance from the type implemented by *Parent*. Interestingly, *ChildType* is not a subtype of *ParentType*, since the argument type `MyType` of *plok* is covariantly specialized, meaning that it varies in the same direction as the alleged subtype relation[17]. Although not in a subtype relationship, *ChildType* can be derived from *ParentType* by type inheritance.

More generally, an interface S can be derived by type inheritance from another interface T, but S does not necessarily contain T [CCHO89]. This issue will be elaborated with greater devotion in chapters 5 and 6, where among other things we will conclude that subtyping and inheritance are two independent notions and induce distinct subtyping and inheritance hierarchies [CCHO89]—subtyping is not inheritance [AL90, CHC90].

[17] `MyType` occurring as the result type of a method does not prevent a subtype relation between the object types, because `MyType` is covariantly specialized and complies with the rules for function subtyping [Ame90].

Some studies [KMN+90] use the term "conformance" to denote substitutability and mistakenly use the term "subtyping" instead of type inheritance (which is restricted to maintain conformance). "Subtyping" is referred to as a conceptual economy for incremental design. The important point to note is that, in contrast to [KMN+90], subtyping does not necessarily have to rely on explicit derivation [KLMM94].

- The language POOL-I [AL90] employs structural subtyping between fully specified types—types in POOL-I are defined from scratch without being based on other types. POOL-I separates the notions of type and implementation and has distinct subtype and implementation hierarchies. Type inheritance is coupled to implementation inheritance: since every implementation implicitly defines a type, a subimplementation defines an inherited type. An implementation may refer to the type associated with `self` via the keyword `MYTYPE`, which covariantly changes its meaning in subimplementations.

- Eiffel [Mey97] offers a similar form of type inheritance, denoted by the use of "like Current" (also known as "declaration by association" [Mey88]) which is covariantly specialized in heirs to refer to their type. Unfortunately, this approach may produce "subtype" instances which are actually not substitutable for supertype instances [CCHO89]: since Eiffel identifies types with implementations, the covariantly specialized implementations are regarded as subtypes, clearly breaking the subtyping rules for object types.

- RAPIDE [KLMM94] employs structural subtyping between interfaces that may be derived by type inheritance—RAPIDE has an explicitly derived type inheritance hierarchy and a separate, structurally inferred subtyping hierarchy [KLMM94].

- In chapters 2 and 3, all the examples that demonstrated named subtype relationships used types that were specified without inheritance. This illustrated that fully specified types are not only meaningful with structural but also with named subtyping.

- [BR97] introduce *signatures* into C++ as a language extension for abstracting types and decoupling subtyping from inheritance—the signature hierarchy may be implemented independently from the implementation hierarchy. A signature specifies the member functions that any implementation of the signature needs to have. The subtype relationships between signatures and the conformance relationships between signatures and implementations are statically inferred by the compiler. In addition, signatures may inherit from other signatures, leading to a type inheritance hierarchy distinct from the structurally inferred subtyping hierarchy.

Nevertheless, for reasons of conceptual economy and simplicity, many languages based on named subtyping [KMN+90, DGLM95] collapse the notions of subtyping and type inheritance into a single mechanism. In particular, this is true for object-oriented languages such as C++ [Str97] and Eiffel [Mey97] that unify subtyping and inheritance.

Type inheritance shares the common preferences of inheritance (Section 4.1), namely the reduction of program size and the improvement of flexibility and maintainability—changes to one type declaration are propagated to all type declarations inheriting from it. If used carefully, then type inheritance (and inheritance in general) can become an efficient means to maintain consistency and integrity of code that is evolving over time [KLMM94].

Chapter 5

Subtyping vs. Inheritance

In sections 2.2.4 and 3.1 we introduce object types and implementations as separate and autonomous notions related solely by the implements relationship. Sections 2.2, 4.4, and 4.3 describe subtyping, type inheritance, and implementation inheritance as moderately independent mechanisms concerned with the manipulation of types and implementations, respectively.

This unequivocal approach to the separation of subtyping and inheritance has so far been accepted beyond reproach—the reasons for a clear distinction are widely recognized and acknowledged in contemporary research [Sny86, CHC90, Har91, Hau93, Tai96, Sou97, Roy98] and practice [AL90, KMN+90, LP91, Por92]. Recently, the Java programming language[1] [GJS96] has made the dichotomy accessible to a broader, more commercially-oriented audience, receiving remarkable approval in the programming community and massively gaining on popularity in commercial applications [Bro98].

Nevertheless, a majority of the modern object-oriented programming languages like C++ [Str97] and Eiffel [Mey97] have opted for the unification of the notions of type and implementation into a single concept. Subsequently, subtyping and inheritance are collapsed into a combined mechanism, meaning that the formerly separate and independent subtype and inheritance hierarchies coincide in a single hierarchy expressing both relationships.

This chapter will give insights into the preferences and weaknesses of identifying subtyping with inheritance. Later we will seek the road back to where we started, returning to the separation of the concepts. We will exhaustively discuss both approaches, illustrating the differences between subtyping and inheritance by means of concrete examples. Finally, we shall conclude that the identification of subtyping and inheritance severely compromises the expressiveness of strongly-typed object-oriented programming [CCHO89].

5.1 Identification of Types and Implementations

The precise distinction between object types and object implementations is not quite so generally acknowledged beyond controversy, finding not only proponents in literature. For example, in [Rum93] the separation between types, seen as "logical black box specifications of externally visible behavior", and implementations, seen as "descriptions of internal form

[1] Java [GJS96] is a new object-oriented programming language with separate linguistic constructs for defining interfaces and implementations.

used for convenience without regard to logical typing", is rejected as dangerous and unproductive. The terms type and implementation are freely interchanged and used synonymously [HB87]. Taking a similarly extreme position, [Mey88, Mey90] unify the apparently distinct programming notions of modules and types, regarding this identification as being central to object-oriented software architecture and giving object-oriented programming languages and design their strong conceptual integrity.

5.1.1 Classes

Following the previous line of thought, many strongly-typed object-oriented programming languages such as C++ [Str97], Eiffel [Mey97], Trellis/Owl [SCB+86, HB87], and Simula [DN66] equate types with implementations [Ped89, CCHO89, CHC90, Hau93, Tai96, Roy98], meaning that the independently introduced notions of object types and implementations are combined into a single language construct—the *class* [Mey88, Sin95]. Class specifications are used to express both a concrete object type and simultaneously the corresponding implementation. In this sense, the class construct covers behavioral as well as implementational aspects of objects[2] [Por92].

Classes type their instances [Cas95], meaning that types can no longer be defined independently from concrete implementations. All instances of a class must have the same type [PS92] and all objects belonging to a specified type have the same implementation—class declarations *are* type declarations [WZ88, CCHO89]. Therefore, an object's type identifies that object's internal representation and implementation [CCHO89].

The identification of types and implementations in class-based languages is typically realized by automatically associating a type system with the set of specified classes [WZ88, DT92]—an appropriately chosen type is assigned to each class. A class C implements its associated type σ, meaning that all instances of class C belong to the type σ.

This coupling of types and implementations may seem to contradict our definition of types (Section 2.1), which stated that a type should, in principle, not provide any information about the implementation of its elements. However, we can reconcile this definition of types with the identification of types and implementations by reinterpreting the predicate associated with the type of the class. The predicate is redefined to characterize the set of all potential class instances. The type predicate associated with a class C is defined as "is an instance of class C" [WZ88].

Example 5.1: Unification of interfaces and implementations into classes.
Consider the following interface *Bag* and a possible implementation of its objects. The implementation *BagImpl* realizes the required functionality by redirecting the messages to an integer stack (not defined here):

> `interface` *Bag*
>   *put*(*int*) `returns nothing`
>   *get*() `returns` *int*
> `end` *Bag*

[2] Smalltalk-80 uses the term *class* to focus on implementations and *subclassing* to denote the inheritance of implementations [GR83, Por92]. Types and subtyping in such dynamically-typed languages are discussed in sections 2.1 and 5.3.2.

```
implementation BagImpl implements Bag
    var contents : Stack := new StackImpl
    method put(x : int) returns nothing
        contents.push(x)
    method get() returns int
        return contents.pop()
end BagImpl
```

By merging these separate definitions, the interface *Bag* and its implementation *BagImpl* can be replaced by a single class (defined in terms of the unspecified class *Stack*):

```
class Bag
    var contents : Stack := new Stack
    method put(x : int) returns nothing
        contents.push(x)
    method get() returns int
        return contents.pop()
end Bag
```

The class *Bag* defines both the object type, consisting of the messages *put* and *get*, and the implementation of all its elements. This definition is shorter and more readable.

5.1.2 Identification of Subtyping and Inheritance

The consequence of unifying types and implementations into classes is to collapse subtyping and inheritance into a single language mechanism applicable to classes [CHC90, Roy98]. This can be accomplished by allowing class specifications to inherit from other class specifications and by automatically superimposing a subtype hierarchy onto this class inheritance hierarchy [Ped89, DT92]. The inheritance hierarchy serves as the basis for the type system [Mey90]. The unified hierarchy determines both the subtype relationships and all applications of code reuse via inheritance [Por92], both roles being handled by a single mechanism [LP91]. Consequently, the notions of interface containment (Section 2.2.6) and inheritance are confused [CCHO89]. We will use the terms *subclass* and *superclass* to denote the relationships in the unified hierarchy.

Strongly-typed object-oriented programming languages like C++ and Eiffel merge the inheritance and subtype relationships [Por92, Sin95], insisting that subimplementations generate subtypes [Ghe91, BCC+96]: the type of a subclass C must be a subtype of the type of C's superclass. The type checking associated with the superimposed type system determines whether the given class hierarchy conforms to the semantics of the type system [DT92].

In class-based languages, the conformance rules become very simple [Ped89]: an instance of a class C' conforms to an instance of a class C (which is equivalent to the corresponding types being in a subtype relationship) if C and C' are the same class or if C' is a descendant of C in the class hierarchy. By virtue of the inheritance mechanism and the superimposed type system, instances of descendants of a class C are guaranteed to support all methods defined by C. We refer to this as *class-based conformance*.

In fact, it has been argued that class-based conformance is the only reasonable approach to designing hierarchies, in terms of modularity and maintenance [Rum93]. In particular, [Rum93] altogether denounces implementation inheritance decoupled from subtyping and proposes that a subimplementation should always be completely compatible with its superimplementation in every respect and context. This accounts to the already elucidated equation of implementations and types.

The class hierarchy represents the basis for checking the legality of assignment statements and parameter passing [BR97]. In contrast to systems which separate types from implementations (sections 2.1 and 2.2.2), in class-based languages the variables are constrained by classes and the subclass relationships determine the legality of assignment statements [Mey88]: the basic constraint on assignment and parameter passing is that the source object's class must be a descendant of the class constraining the target variable [Mey90, Mey92]. This approach is type-safe because the coupling of subtyping and inheritance means that subclass objects are substitutable for the corresponding superclass objects—subclasses generate subtypes [Jon92, BCC+96].

Example 5.2: Unification of subtyping and inheritance.
The class *Bag* from Example 5.1 can be used for defining a more sophisticated type of bag:

```
class ComplexBag subclass of Bag
    method print() returns nothing
        contents.print()
    method newEmpty() returns ComplexBag
        return new ComplexBag
end ComplexBag
```

The class *ComplexBag* inherits the specifications from *Bag*. At the same time, the type associated with *ComplexBag* is a subtype of the type associated with *Bag*. Thus, the subclass is a subtype, meaning that instances of *ComplexBag* can be used in any context expecting instances of *Bag*. Note that this substitutability of subclass objects for superclass objects is not merely a coincidence, but an integral part and consequence of class-based conformance. A subclass declaration whose associated type is not a subtype of the type associated with the superclass is rejected as illegal.

The following section will illustrate the particular hindrances and intricacies pertaining to the identification of types with implementations (or subtyping with inheritance) by means of concrete examples, examining to what extent the entailed restrictions have practical impacts on the expressiveness of a language. We will conclude that the distinction between types and implementations (especially the presence of separate linguistic constructs) contributes to the flexibility and expressiveness of object-oriented programming [CCHO89, BR97].

5.2 Restrictions on Subtyping and Inheritance

A considerable share of the problems adhering to common strongly-typed languages stems from the multipurpose nature of the class construct: in many object-oriented programming languages the class and inheritance mechanisms are both heavily overloaded, each embracing a multitude of different (and potentially incompatible) functionalities [BHJL86, LP91, BR97]. Classes typically experience the following employments:

- A class C defines an object type σ [BR97], meaning that every instance of class C is an element of type σ.
- A class C with object type σ defines the implementation for all objects belonging to type σ [BR97]—all elements of σ necessarily share the implementation specified by C.
- A class typically acts as a repository for the methods of its instances [BHJL86], typically in terms of some static textual collection of the instance methods' source codes and possibly as some form of a run-time repository for the methods' implementations.
- Classes serve as templates from which objects are generated, describing how these class instances are to be built [BHJL86, DT88, WZ88].
- Classes provide an intuitively appealing level of granularity as the unit of code reuse [Mey90], essentially constituting the basis for inheritance and substitutability relationships [BHJL86, DT88, BR97].

As shall soon become apparent, the coupling of subtyping and inheritance has immediate and far-reaching implications in terms of the expressiveness and flexibility of subtyping and inheritance [Sny86, CCHO89, CHC90, LP91, Kil92]: the inheritance hierarchy is restricted to coincide with the subtyping hierarchy, and the subtyping hierarchy is restricted to coincide with the inheritance hierarchy [Por92].

Inheritance is restricted to subtyping: Type-safety is enforced by restricting inheritance between the associated implementations to maintain the superimposed subtype relationship [CHC90, Kil92, Tai96]. Assuming that class C is a subclass of class C' and that σ and τ are their associated types, respectively, then type σ must be a subtype of type τ.

Forcing inheritance to preserve the associated subtype relations precludes certain possibilities of code reuse even when substitutability is not an issue [Kil92]. More specifically, this curtails the applicability of inheritance by forbidding the cancellation (exclusion) of public methods [Jon92, BR97] and the arbitrary redefinition of method signatures and implementations. Overriding must respect subtyping [HB87]. Inheritance for code sharing is restricted to situations where there is also a subtype relationship.

Example 5.3: Inheritance is restricted to subtyping.
A fat bag is a bag which only grows, while a normal bag may grow and shrink [LW93]. Clearly, fat bags are not substitutable for normal bags. Assuming implementation inheritance were decoupled from subtyping, a fat integer bag could be based on the definition of *Bag* (from Example 5.1) as follows:

```
class FatBag subclass of Bag
    exclude get
    method iselement(x : int) returns bool
        return contents.iselement(x)
end FatBag
```

The specification of *Bag* would be used purely for convenience, allowing arbitrary cancellation or redefinition of methods. Unfortunately, since inheritance is bound to subtyping, this appealing definition of *FatBag* becomes illegal because the type associated with *FatBag* is not a subtype of the type associated with *Bag*.

Most object-oriented languages allow existing method bodies to be overridden in descendants[3]. However, compelling the descendants to generate subtypes restricts how the associated object types may be respecified in the descendant classes [HB87]. The superimposed type system allows only "safe" redefinitions, meaning such redefinitions that cannot lead to run-time type errors by violating the substitutability principle [Tai96]. The subclass objects can be used anywhere the superclass objects can be used if the new method is applicable to everything the overridden method applied to, meaning that the new method's type must be a subtype of the overridden method's type [Har91, BC96]. The concrete approach varies strongly from language to language:

- Trellis/Owl [SCB+86] guarantees type-safety by allowing functions to be overridden according to the rules for function subtypes [CHC90], namely contravariant argument types and covariant result types. This approach offers a maximum degree of flexibility while retaining complete compile-time type checking [HB87].

- The language C++ [Str97] has no function subtyping at all [CHC90, Har91], meaning that the argument and result types of a method may not change in the descendant class[4]. Subtyping in C++ is *invariant* [BCC+96].

- Trading type-safety for increased flexibility, Eiffel [Mey97] actually allows argument types to be *covariantly specialized* in the subclass methods [CHC90, BCC+96]. Therefore, the objects of a subclass may not necessarily be substitutable for the objects of the superclass—subclasses don't always generate subtypes [BCC+96]. Nevertheless, Eiffel preserves the claim that subclasses generate subtypes [BCC+96]. To compensate for the resulting insecurity in the type system, most Eiffel compilers perform a link-time data-flow analysis of the program (also known as *system validity check* [Jon92, BC96, BCC+96]) to catch possible run-time type errors[5]. In a sense, the term "subtype" loses its clear meaning in Eiffel, since objects of alleged subtypes cannot be used in every context where supertype objects are expected [BCC+96].

- The lack of compile-time type constraints on formal parameters in Smalltalk-80 [GR83] may raise the conjecture that methods may be arbitrary overridden. Section 5.3.2 will disprove this superficial assumption, showing how such rashness may lead to run-time type errors if subclasses are falsely regarded as subtypes.

[3]The notion of behavioral subtyping [Wing87, LW94] forbids the arbitrary reimplementation of methods, restricting the subtype's implementation to satisfy certain behavioral constraints.

[4]In fact the situation in C++ is somewhat more intricate. Assume there exists a same-named method in both the superclass and subclass specification. The definition in the subclass overrides the definition in the superclass only if their function types are exactly identical. Otherwise the subclass method *overloads* the superclass method and the inherited method still exists in the subclass: the two methods exist in parallel and are regarded as completely unrelated methods which incidentally have the same name—a dangerous source of subtle errors [Har91].

[5]One problem with this approach is that the system validity check must be conservative to some degree, potentially rejecting programs that are actually type-safe.

Example 5.4: Overriding must respect subtyping.
Assume the definition of some class A whose associated object type is recursive in the argument type of the message *foo*:

```
class A
     method foo(x : A) returns nothing
end A
```

Consider the following definition of some descendant of A:

```
class B subclass of A
     method foo(x : B) returns nothing
end B
```

The specification of B overrides the definition of *foo*, changing the argument type to denote the type associated with B. Due to the subtyping rules for function and object types (sections 2.2.3 and 2.2.4), the subclass does not generate a subtype of the superclass. The type associated with B is not a subtype of the type associated with A. We reject the definition of B as illegal, since it does not satisfy the type checking semantics of the superimposed subtype hierarchy[6].

It is important to note that the loss of subtyping is not the consequence of some peculiar problem with the definition of function subtyping [BCC+96]. On the contrary, Section 6.1 will demonstrate how definitions such as B in the prior example would lead to run-time type errors if they were not disallowed by the compiler.

To summarize, the area of applicability of implementation inheritance is diminished and surrenders its predominance as a mechanism solely for the use of implementors, which were allowed to use inheritance for mere convenience. Implementation inheritance was a linguistic concept that allowed programs to be built in some particular way and could be employed (but was not necessarily limited) to the implementation of subtype objects [Lis87, Cook92, Tai96]. However, by coupling inheritance to subtyping, inheritance is made visible to the clients via the external interface of the class [Sny86].

Subtyping is restricted to inheritance: A subtype's objects must share a common representation and implementation with the supertype's objects. The class associated with the subtype must inherit the instance variables and methods from the class associated with the supertype—the subtype relationship is based on the inheritance relationship [CHC90, Mey90].

Since it is nonsensical to infer inheritance relationships, the unified class hierarchy must be defined explicitly. Therefore, the corresponding subtype relationships are also explicitly defined rather than inferred by the system. Subtyping cannot be structural. Subsequently, a type can only be a subtype of another type if the associated classes are also related by inheritance. Practically, this has the following consequences:

1. A type cannot have several different implementations[7]. This results directly from the identification of types and implementations and means that all elements of a type must

[6]The definition would be accepted by Eiffel compilers [Mey97], which would typically attempt to catch suspicious applications (in terms of potentially leading to run-time type errors) at link-time.

[7]Section 5.3.1 will attempt to alleviate this restriction by means of *abstract classes*.

have the same implementation. Different implementations for this type may not coexist and are not substitutable for one another in a single system [PS92].

2. The elements of a subtype must share the implementation with the elements of the supertype, meaning that a subtype's objects cannot be implemented completely differently from the corresponding supertype's objects.

Example 5.5: Subtyping is restricted to inheritance.
A class *Matrix* can only provide one single implementation for all matrix objects. However, different implementations with different representations and specialized algorithms may actually be required, for example optimized representations and algorithms for sparse and dense matrices, respectively. Similarly, the class *EfficientIntBag* may want to implement a subtype of *IntBag* but provide additional functionality and a completely different, more efficient internal implementation. Unfortunately, both cases are precluded by equating types with classes and subtyping with inheritance.

Before we return to the separation of subtyping and inheritance in the next section, note that although unifying inheritance and subtyping is usually entailed in a loss of flexibility and expressiveness, it does represent a well-tailored and successful economy of concepts. The language is simplified and the combined hierarchy is fully sufficient for a multitude of applications and hierarchies. In addition, the unification has significant implementational advantages in strongly-typed object-oriented programming languages [Sny86].

5.3 Separating Subtyping and Inheritance

As stated by [LP91], one of the fundamental problems with most strongly-typed object-oriented programming languages is the dual role played by the inheritance mechanism. However, the two relationships are in fact clearly distinguished concepts because they deal with different aspects of programming [AL90]. In general, subtyping and inheritance are incompatible with each other and represent distinct or even opposing views of the same program [LP91, Cook92]:

1. Implementation inheritance is a linguistic concept concerned with the sharing of algorithms and representations. In this sense, implementation inheritance should be solely of interest to object implementors.

2. Subtyping is a substitutability relationship between object behaviors, describing when an object of one type may be type-safely substituted for an object of another type. This concern for substitutability of object behaviors should make subtyping primarily of interest to object users.

Inheritance supports reusability for the implementor while subtyping supports reusability for the client [LP91, Tai96]. There is a wide consensus that inheritance is a producer's mechanism [Cook92], being fairly independent of a client's view or use of objects (expressed by subtyping). Nevertheless, both concepts are commonly referred to as "inheritance", prolonging a long tradition of misinterpretation and misuse of terminology [PS92]. The term "inheritance" adopts a variety of different semantics in both research and practice:

- [Car88] identifies the term "inheritance" with subtype relationships between record types, his notion of inheritance having absolutely nothing in common with code sharing. Similarly, the term inheritance is used in [CL95] to denote subtyping.
- "Classes" in Smalltalk-80 [GR83] are understood purely as object implementations and inheritance as a mechanism facilitating code sharing between these classes. In particular, types and subtype hierarchies are independent of the class hierarchy (see Section 5.3.2).
- POOL-I [AL90] clearly separates between types and implementations. Inheritance is viewed purely as a low-level mechanism for reusing existing implementations. Implementation hierarchies are completely independent from types and serve to categorize implementations according to their shared definitions.
- In C++ [Str97], the term inheritance (or "derivation") is used to denote the combined subtype/inheritance relationship. Many tutorials and introductory texts on C++ [Str87, Eck95], and tutorials on object-oriented programming in general, present only a very vague distinction between object behavior and implementation, or fail to clarify the issue altogether.
- In contrast to C++, RAPIDE [KLMM94] has distinct hierarchies for implementation inheritance, type inheritance, and structural subtyping. This approach has proven successful because these hierarchies are frequently used independently and, in general, do not coincide.
- [Weg88] uses the term inheritance to refer to the reuse of code and to the formal behavioral relationship between types.

Most of these studies confine themselves to only one specific viewpoint of inheritance, ignoring other potential viewpoints. The discrepancy is highlighted by [Tai96], who uses the term inheritance to denote both a programming language mechanism and a facility for conceptual modeling in software development. "Inheritance" is employed in the context of

- the low-level implementation phase, where inheritance supports the reuse of existing classes as the basis for defining new classes,
- and as part of the high-level design phase, where inheritance serves to model the application domain in terms of specialization relationships [Weg87].

Basically, [Tai96] concludes that "inheritance" should be used to denote an implementation technique bearing no direct resemblance to conceptual modeling. A clear distinction is necessary between inheritance as a low-level mechanism for reusing existing code and subtyping as a mechanism for classifying object behaviors. In a similar vein, [Ame90, Por92, Hau93, BR97] proclaim the necessity of a type abstraction mechanism that is independent of implementations. The orthogonality of subtyping and inheritance requires a clean separation of types and implementations[8] [LP91, Kil92].

[8] Some languages separate types from implementations, but fail to clearly distinguish subtyping from inheritance. For example, Guide [KMN+90] and Java [GJS96] have distinct constructs for defining types and implementations. Nevertheless, the implementation hierarchy is coupled to the subtyping hierarchy, meaning that code may only be shared between implementations that induce common behavior. Inheritance is still bound to subtyping.

In many cases, the specific differences between subtyping and inheritance may not be immediately visible in practice [LP91]. Commonly the implementation hierarchy coincides completely with the subtyping hierarchy [Ame87]. Indeed, there often exists a one-to-one correspondence between types and their implementations [Ame87, Por92], meaning that each type has exactly one implementation.

Example 5.6: Identical type and implementation hierarchies.
For example, take into consideration the following implementations and their associated, most specific object types:

```
implementation Vehicle(speed : int)
    var speed : int := speed
    method speed() returns int
        return speed
end Vehicle

implementation Car(speed : int, fuel : string) inherits Vehicle
    var fuel : string := fuel
    method fuel() returns string
        return fuel
end Car

implementation Horse(speed : int, breed : string) inherits Vehicle
    var breed : string := breed
    name name : string := "Bob"
    method breed() returns string
        return breed
    method name() returns string
        return name
    method name(s : string) returns nothing
        name := s
end Horse
```

$$
\begin{aligned}
&\mathit{Vehicle} \equiv \langle\langle \mathit{speed} : \mathit{unit} \to \mathit{int} \rangle\rangle \\
&\mathit{Car} \equiv \langle\langle \mathit{speed} : \mathit{unit} \to \mathit{int}; \mathit{fuel} : \mathit{unit} \to \mathit{string} \rangle\rangle \\
&\mathit{Horse} \equiv \langle\langle \mathit{speed} : \mathit{unit} \to \mathit{int}; \mathit{breed} : \mathit{unit} \to \mathit{string}; \mathit{name} : \mathit{unit} \to \mathit{string}; \\
&\qquad\qquad \mathit{name} : \mathit{string} \to \mathit{unit} \rangle\rangle
\end{aligned}
$$

The types defined inherently by *Vehicle*, *Car* and *Horse* are in subtype relationships which incidentally coincide with the specified inheritance relationships: the implementations *Car* and *Horse* inherit from *Vehicle* and the types *Car* and *Horse* are subtypes of *Vehicle*.

In the prior example, the type defined by the subimplementation is a subtype of the type defined by the superimplementation. These types and implementations could just as well be defined in terms of classes, the subclass relationship replacing the inheritance and subtype relationships. Inheritance suggests that when a class C inherits from a class C', then each instance of C will have at least all the variables and methods that C''s instances have

[Ame87]. Therefore, objects of class C may be viewed as being substitutable for objects of class C' [Ame90].

Seemingly, one may be tempted to regard the hierarchical relationships induced by inheritance and subtyping as always being isomorphic [Ame90, Por92]. The frequent congruity between types and implementations would intuitively make it justified to merge the formerly independent relationships into a single one. This view has obstinately prevailed in the object-oriented community [Ame90].

Nevertheless, lately it has been widely recognized that the identification of subtyping and inheritance may actually prove too stringent in practical programming, entailing a number of disadvantages [Mey88, Ame90, CHC90, DT92, Hau93, Tai96, Sou97, Roy98]. In particular, the hierarchies of object behaviors and the hierarchies of implementations may be in conflict and difficult to reconcile with each other [BR97]. The attempt to unify these two hierarchies will inevitably be problematic.

[DT92] claim that superimposing a subtype hierarchy onto the inheritance hierarchy leads to more confusing or even "wrong" type or inheritance relationships. For example, classical misuses[9] of inheritance are unintentionally demonstrated in [AGS91, Mey97]:

- The class *FixedList* is defined in terms of multiple inheritance by inheriting from the classes *Array* and *List*.
- Similarly, the class *FixedStack* is defined in terms of multiple inheritance by inheriting from *Stack* and *Array*.
- The class *Stack* is defined by inheriting from *List*.

Evidently, this use of inheritance is highly questionable in class-based languages, since an inheritance relationship automatically implies a substitutability (subtype) relationship. Hence, it may be strongly doubted whether fixed lists or fixed stacks are substitutable for arrays. Many operations applicable to arrays are, in general, not applicable to lists or stacks [Tai96]. Deriving *Stack* from *List* in a class-based system is even more paradox, since the substitutability relationship will typically be the other way around: lists are substitutable for stacks. Decoupling inheritance from subtyping alleviates these problems by allowing *Stack* to inherit from *Array* without implying that stacks are substitutable for arrays[10] [AL90].

In general, the exact one-to-one correspondence between the type and implementation hierarchies is not given [Por92], even though the differences may remain in obscurity for a wide variety of applications. However, a number of more subtle examples reveals that the distinction between interfaces and implementations is much more intricate than commonly assumed. Consequently, the discrepancy between subtyping and inheritance crystallizes [Ame87].

Separating types from implementations allows a single type to have multiple different implementations, contrasting the approach taken with abstract data types [LSAS77] or class-based languages such as C++ [Str97]. For example, both *DenseMatrix* and *SparseMatrix*

[9] Other questionable uses of inheritance (both accidental and deliberate) may be found in [Moon86, HB87, Rum93, Sin95].

[10] Although the subtyping issue is solved, we feel that such uses of implementation inheritance should be avoided: *Stack* should use *Array* as a component rather than inherit from it [Rum93].

may implement the object type *Matrix* [Por92]. As elaborated in Section 3.2, a single implementation may implement more than one type and a single type may be implemented by more than one implementation [BHJL86, PS92].

It proves convenient to decouple the concepts of subtyping and inheritance and to consider each on its own right [Ame87, DT88, Ame90, BR97]. This effectively furnishes the programmer with the possibility to employ subtyping without being bound to inheritance and to use inheritance without being restricted to subtyping [Cha92]. Hence, the implementation hierarchy is only related to the subtype hierarchy in terms of the implements relationship. Since inheritance is not a sound basis for the subtype relationship [CHC90], the separation of the notions may solve some of the problems traditionally adhering to object-oriented programming [AL90], such as the inconsistent behaviors resulting from overriding methods in subclasses [DT88]. We return to this issue in Chapter 6.

Subtyping without inheritance: Assume an implementation I that implements an object type σ. Another implementation I' may implement an object type τ, which is a subtype of type σ ($\tau \leqq \sigma$). However, in contrast to class-based systems, the implementation I' is not bound to inherit from the implementation I, possibly employing a completely different internal structure and different algorithms. The implementation of the subtype is not necessarily related to the implementation of the supertype—subtyping is not bound to inheritance [Sny86].

In other words, the sharing of implementation code is not the only possibility to realize substitutability [Ame90]. A specific functionality can be obtained by means of different representations and algorithms [Horn87]. Therefore, the implementation of a subtype is not stringently compelled to inherit the instance variable declarations and method definitions from the implementation of its supertype. Since substitutability is concerned only with an object's externally observable behavior and is independent of internal details, the objects of a subtype are still perfectly substitutable for the objects of the corresponding supertype, even though they do not share a common implementation. In essence, substitutability can be achieved without code sharing.

Example 5.7: Subtyping without inheritance.
A stack may be implemented in terms of a list, or, perhaps more efficiently, in terms of an array [Ame90]. Consider the following interfaces:

```
interface Stack
    push(int) returns nothing
    pop() returns int
end Stack

interface Stack2 inherits Stack subtype of Stack
    dup() returns nothing
end Stack2
```

The interface *Stack2* is obtained by inheriting the signature definitions *push* and *pop* from the interface *Stack* and complementing them with the additional message *dup*[11]. Further,

[11] The typical semantics of *dup* is to duplicate the topmost stack element.

the declaration of *Stack2* explicitly states itself to be a subtype of *Stack* (named subtyping). Interestingly, while this example will demonstrate subtype and implementation hierarchies that do not coincide, the interface conformance hierarchy and the interface inheritance hierarchy do coincide. The interfaces *Stack* and *Stack2* are in a subtype *and* type inheritance relationship. Assuming the presence of predefined integer lists and arrays, the interfaces may be implemented as:

```
implementation StackImpl implements Stack
    var contents : List := new List
    method push(x : int) returns nothing
        contents := contents.prepend(x)
    method pop() returns int
        var temp : int := contents.head()
        contents := contents.tail()
        return temp
end StackImpl

implementation Stack2Impl implements Stack2
    var contents : Array := new ArrayImpl(100)
    var index : int := 1
    method push(x : int) returns nothing
        if index > contents.size()
            contents.expand(100)
        contents.put(index,x)
        index := index + 1
    method pop() returns int
        var temp : int := contents.at(index - 1)
        index := index - 1
        return temp
    method dup() returns nothing
        var temp : int := self.pop()
        self.push(temp)
        self.push(temp)
end Stack2Impl
```

The implementations *StackImpl* and *Stack2Impl* implement the interfaces *Stack* and *Stack2*, respectively. While the types are in a subtype relationship, the corresponding implementations are coded independently and without inheritance, employing distinct internal representations and algorithms.

Inheritance without subtyping: Assume an implementation I that implements an object type σ. Another implementation I' inherits the instance variable declarations and method definitions from I and implements an object type τ. In contrast to class-based systems, where the type τ is required to be a subtype of type σ and this superimposed subtype relationship is enforced and checked by the compiler[12], the types σ and τ need

[12] C++ and Eiffel slightly relax this requirement: C++ [Str97] allows *private* inheritance, meaning that

not be related by subtyping. The type implemented by a subimplementation is not necessarily related to the type implemented by the superimplementation [CHC90]—inheritance is not bound to subtyping.

Thus, code reuse realized in terms of inheritance is not restricted to automatically imply the substitutability of the implemented object behaviors [Ame90]. Discharging the subimplementation from producing subtypes allows the arbitrary redefinition and exclusion of inherited methods, without regard to the constraints described in Section 5.2. Unrelated types may have related implementations [Por92].

In class-based systems, the definition of an implementation is always automatically accompanied by the definition of an object type. This type plays a profoundly significant role in the process of inheritance. Relieving the inheritance mechanism from this unnecessary limitation facilitates the introduction of *inheritance packages* [Ame87], solely for the purpose of code sharing [DT92], in terms of partial implementations or *mix-ins*[13] [Por92]. There is no guarantee that the newly-derived implementation will be a subtype of the initial implementation [Cus91].

Example 5.8: Inheritance without subtyping.
In the face of decoupled subtype and inheritance hierarchies, the inheritance mechanism can again be used purely as an implementation technique. The problem from Example 5.3 can now be correctly coded as:

```
interface FatBag
    put(int) returns nothing
    iselement(int) returns bool
end FatBag

implementation FatBagImpl inherits BagImpl implements FatBag
    exclude get
    method iselement(x : int) returns bool
        return contents.iselement(x)
end FatBagImpl
```

The implementation *FatBagImpl* inherits the instance variable declarations and method definitions from the implementation *BagImpl* (defined in Example 5.1), excluding the method *get* and defining an additional method *iselement*. Therefore, the type *FatBag* implemented by *FatBagImpl* is not in any sort of subtype relationship with the type *Bag* implemented by *BagImpl*.

Example 5.9: Subtyping and inheritance hierarchies in opposite directions.
A frequently stressed example [Sny86, Por92, KLMM94, BR97] demonstrates the rather extreme case in which the independent subtype and implementation inheritance relationships run in opposite directions to one another:

public functions become private in the subclass, and Eiffel [Mey97] allows a descendant to exclude features formerly exported by its ancestor. In both cases the substitutability between ancestor and descendant instances is lost. C++ copes with the situation by breaking the static subtyping chain and Eiffel employs a link-time system validity check [TW95].

[13] Mix-ins are units of implementation which do not represent complete implementations on their own but are created with the intention to be "mixed" into other implementations [Hen86, BC90].

```
interface Queue
    put(int) returns nothing
    get() returns int
end Queue

interface DeQueue inherits Queue subtype of Queue
    putback(int) returns nothing
    getback() returns int
end Queue
```

The interface *DeQueue*[14] inherits the signature definitions from *Queue* and explicitly states itself to be a subtype of *Queue*. The interfaces *DeQueue* and *Queue* are in parallel subtype and type inheritance relationships. To simplify and shorten the further presentation, we elide the method bodies of the implementation *DeQueue* (which could be implemented in terms of a doubly-linked list):

```
implementation DeQueueImpl implements DeQueue
    var contents : DList := new DList
    method put(x : int) returns nothing
    method get() returns int
    method putback(x : int) returns nothing
    method getback() returns int
end DeQueueImpl

implementation QueueImpl inherits DeQueueImpl implements Queue
    exclude putback, getback
end QueueImpl
```

Since *DeQueueImpl* already provides all the functionality required by *QueueImpl*, the implementation of *Queue* is obtained by simply inheriting the code from *DeQueueImpl* and removing the superfluous methods *putback* and *getback*. The implementation inheritance hierarchy runs in exactly the opposite direction of the subtype hierarchy: *QueueImpl* inherits from *DeQueueImpl* and *DeQueue* is a subtype of *Queue*. Note that implementing *DeQueue* by inheriting from *QueueImpl* is generally not possible, because the implementation of *Queue* will typically use a representation (such as a singly-linked list) which only allows elements to be added and removed from one single side.

Subtyping and inheritance are distinct concepts, lying on different levels of abstraction in a system. They should not be confused in any object-oriented programming language [Ame87]. Apart from these two relationships, we have also discussed other relationships between types and classes. To recapitulate:

1. Subtyping is concerned with the externally observable behavior of objects and serves as a higher-level relationship between these behaviors (expressed in terms of types) [Ame87, Tai96]. The subtyping (conformance) rules lead to a type hierarchy based on behavioral specification, describing substitutability relationships between objects.

[14] A double-ended queue allows elements to be added and removed from either side.

2. Inheritance constitutes a convenient mechanism for the sharing of code [Ame87, Tai96, MT97]. New program units are derived by specifying the changes to existing program units [SCC+93], meaning that less code needs to be written and maintained [Por92]. Implementation inheritance is concerned with the internal structure of a type's objects and operates at the level of these object implementations. Implementations are related to each other in the inheritance hierarchy.

3. Type inheritance is concerned with the manipulation and sharing of type definitions. The derived type is not necessarily a subtype of the initial type [ASB94].

4. Implementations and types are related to each other by the implements relationship.

While the dichotomy between subtyping and inheritance is usually present in tutorials on object-oriented design, it mostly vanishes in concrete programming languages. A typical text on object-oriented programming will at most vaguely point out a conceptual distinction between subtyping and inheritance, or, more frequently, fail to address the topic altogether. Not surprisingly, the least programming languages actually provide the necessary linguistic constructs to realize the separation. So far, the separation has only been incorporated in a very small number of programming languages:

- POOL-I was one of the first languages with inheritance and structural subtyping as completely separate language mechanisms [Ame87, AL90, Ame90].

- Portlandish has separate constructs for types and implementations and consequently independent (explicitly named) subtype and implementation hierarchies [Por92].

- RAPIDE has independent hierarchies for implementations, structural subtyping, and explicit type derivation [KLMM94].

- A remarkable exception in the domain of commercially successful languages is the recently introduced Java programming language [GJS96], which has separate constructs for defining interfaces and implementations independently. Classes provide object implementations for explicitly declared interfaces. A class may "extend" only one class (single inheritance for implementations), while an interface may extend multiple interfaces (multiple subtyping) [DE97]. Nevertheless, implementation inheritance in Java is still restricted to subtyping.

- Traditional class-based languages such as C++ [Str97] and Eiffel [Mey97] have introduced abstract (or *deferred* [Mey90]) classes as a means to simulate the separation (Section 5.3.1).

- Lately, [BR97] have proposed the introduction of a new type definition construct into C++, namely *signatures*, as a language extension for abstracting types and decoupling subtyping from inheritance. Signatures cleanly separate interfaces from implementations by specifying the member functions that any implementation of the signature needs to have. These relationships between signatures and implementations (classes) as well as the subtype relations between signatures are structurally inferred by the compiler at compile-time. The subtyping hierarchy is completely independent from the class hierarchy, facilitating the decoupling of subtyping and inheritance as a real code reuse mechanism. Signatures may inherit from other signatures, building a type inheritance hierarchy (Section 4.4) quite distinct from the structurally inferred subtyping hierarchy. Subtype polymorphism is realized in terms of signature conformance and inheritance need not define subtype relationships any longer [BR97]. Objects are manipulated via signature pointers (the assignment of object pointers to signature pointers is checked for conformance statically) and the class of the referenced object is, in general, unknown. Signatures enhance C++ with function subtyping, allowing the full range of contravariant argument and covariant result types.

The inclusion of separate linguistic constructs to express interfaces, implementations, subtyping, type inheritance, and implementation inheritance into a programming language will inevitably make the resulting programs more complicated and introduce additional intricacies and subtleties into the language. Thus, the drawback of the separation is the increased complexity of the resulting programs [Tai96]. Independent hierarchies within a program may overly complicate the program structure [Kil92]. To justify the employment of separate hierarchies, the following questions must be individually answered [Por92]:

- Will the separation lead to a cleaner and more flexible language design?
- How often will the subtyping and inheritance hierarchies differ in practice?
    1. If they are isomorphic, how big (if any) will the additional burden be compared to a language with unified hierarchies?
    2. If they are not isomorphic, will the increased complexity justify the expressive power of the separate hierarchies?
- Regarding the potential complexities treated by the prior questions, the most important and practical question is whether the programmer productivity will be improved by the separation—do the preferences outweigh the drawbacks?

The separation of subtyping and inheritance in terms of explicit interfaces provides the expressiveness and flexibility of dynamically-typed languages [CCHO89, CHC90] and the benefits of abstract data types [LSAS77, DT92], while retaining the efficiency and type-safety of strongly-typed languages [BR97]. The cleaner language design allows both mechanisms to be applied more effectively in actual programs [Ame90], thus contributing to the flexibility strived for by subtyping [MT97].

5.3.1 Simulation of Separation via Abstract Classes

Class-based programming languages are only equipped with a single construct for defining object types and their associated implementations, and a single unified mechanism for expressing subtype hierarchies and implementation inheritance. The separation of interfaces from implementations is typically confined to languages based on abstract data types such as Ada [Ada83] and CLU [LZ74, LSAS77]. Nevertheless, the effect of separating interfaces from implementations, and thus alleviating the mutual restrictions of subtyping and inheritance, can (to some degree) be simulated in traditionally class-based languages like C++ [Str97] and Eiffel [Mey97] by means of *abstract* classes[15] [Hür94]:

Abstract Class: A class C is *abstract* if it has at least one abstract method. A method is abstract if it is declared or referenced in C but not implemented in the specification of class C[16]. Since classes with abstract methods are not "complete", meaning that their resources are not sufficient to fulfill their obligations towards clients [WZ88], abstract classes cannot be instantiated [HB87, Hür94, Tai96]. In strongly-typed languages, a compile-time check guarantees that objects can only be instantiated from *concrete* (fully specified) classes[17] [HB87].

Abstract methods are called "pure virtual functions" in C++ [Str97]. Functions are declared as pure virtual functions in the class definition by replacing the function body with `=0`. The counterpart in Eiffel are "deferred features" [Mey97], which are declared by replacing the feature body by the keyword `deferred`. While abstract classes are not explicitly supported by the Smalltalk-80 [GR83] language, they are facilitated by dynamic typing: an abstract method is simply not implemented at all or (by convention) defined to generate a run-time type error, indicating that the method must be overridden in subclasses [HB87].

Abstract classes enable the construction of classes whose definitions have purposely been left incomplete, meaning that their methods may refer to attributes that have not been implemented yet [Tai96]. Abstract classes are typically used for two different purposes[18] [Mey90]:

Partial implementation: a partially abstract class with some (but not only) abstract methods provides a partial implementation of some functionality, expecting the rest of the implementation to be provided by inheritors [Tai96]. In this sense, partially abstract classes are an implementation technique for describing packages of incomplete implementations with some variable part which is left to concrete subclasses [HB87]. The packages are further customized by inheritors who provide concrete definitions for the formerly virtual attributes [Mey90].

Interface specification: a fully abstract class consisting only of abstract methods represents an interface [HB87, Sou97] without any implementation. The abstract class defines a type which is implemented by concrete subclasses [Mey90].

Effectively, by virtue of abstract classes, the single ancestor relationship can now be used to express three different relationships, namely subtyping, inheritance, and the implements relationship:

---

[15] The term *deferred* class is commonly used in the context of Eiffel [Mey88, Mey97]. We will retain the terminology used in C++ and generally use the term *abstract* class. Abstract classes should not be confused with abstract types [BHJL86].

[16] More generally, an attribute a which is referenced but not implemented in an entity P is called a *virtual attribute* [WZ88]. The entity P cannot be used before the attribute a is implemented by an inheritor—the inheritor inherits all non-virtual attributes and must implement the inherited virtual attributes. P relies on all its virtual attributes to be implemented by descendants [WZ88].

[17] In Smalltalk-80 [GR83] abstract classes are not instantiated by programming convention [Tai96].

[18] These approaches may also be mixed: an abstract class can define an interface but nevertheless provide default implementations for some of the specified messages [HB87, Sal98].

1. A class hierarchy of fully abstract classes corresponds to a subtyping hierarchy. Since abstract classes contain no implementation, subtyping is not restricted to implementation inheritance. The relationship between two abstract classes encompasses subtyping and type inheritance. To retain subtype polymorphism, variables can be declared to be constrained by abstract classes—at run-time they will refer to instances of concrete subclasses.

2. As before, classes may inherit code from other fully or partially implemented classes, providing the functionality of the ancestor relationship as a mechanism for code reuse.

3. The relationship between a fully abstract class and a fully implemented subclass corresponds to the implements relationship between object types and implementations (Section 3.2). The abstract class represents an abstract type and classes that implement that abstract type must inherit from the abstract class [BR97]. Thus, a separation between interfaces and implementations as in Ada [Ada83] or CLU [LSAS77] is achieved by declaring abstract classes (interfaces) and concrete subclasses (implementations) [Mey88].

In contrast to languages based on abstract data types, the separation through abstract classes allows a single type to have multiple implementations that simultaneously coexist in the system [Mey88, CCHO89]. Any class may inherit from an abstract class, providing an implementation for the interface represented by the abstract class. Note that this type of implements relationship cannot be structural, since a class implementing an abstract class' interface must explicitly state its adherence to the interface by inheriting from the corresponding abstract class [BR97].

5.3.2 Dynamically-Typed Programming Languages

The introduction of interfaces and the distinction between subtyping and inheritance in strongly-typed object-oriented programming is frequently motivated with the goal to achieve some of the flexibility and expressiveness of dynamically-typed object-oriented programming [CCHO89, CHC90, BR97]. To elaborate the issues of subtyping and inheritance in dynamically-typed languages, we will focus our discussion on Smalltalk-80 [GR83], representing *the* prototypical dynamically-typed object-oriented programming language.

Section 2.1 explained that in Smalltalk-80, variables are not explicitly constrained by types, meaning that it is syntactically legal for any object to be assigned to any variable. Nevertheless, we showed that, although not explicitly declared, types implicitly exist, since a programmer will typically make assumptions about the properties of the objects being manipulated via variables and formal parameters [Cook92, Jon92]:

- Classes define implicit types, corresponding to the set of messages to which an instance of the class can respond. In Smalltalk-80, a *protocol* consists of a set of message selectors[19].

- A variable or formal parameter is constrained by an implicit type, corresponding to the set of messages sent to the referenced objects via that variable or formal parameter. This type will typically only be a subset of some implicit object type [Höl93, Jon92].

[19]Protocols are not present as a language construct, but constitute the documentation of a class and exist in the mind of the programmer.

By convention, the implicit type constraints are usually made visible through the use of names for variables and formal parameters that are indicative of their type [LP91]. Type checking in dynamically-typed languages is postponed until run-time. Assuming a variable a refers to an object x, then a type ("message not understood") error may occur if the implicit type of x does not conform to the implicit type constraining a—x would receive a message that it cannot respond to. In other words, the legality of assignments is based on structural conformance verified at run-time.

Classes in Smalltalk-80 are viewed strictly as specifications of object implementations, defining the instance variables and methods of their instances. Classes are organized by inheritance [Cook92, LP91]. The inheritance structure documents the classes, aiding the users in understanding the implementation of a library [Cook92]. The relationship between a class and its subclasses is one of shared implementation: a subclass may redefine the methods it inherits, but necessarily shares the internal representation and typically many methods specified by its superclass [BHJL86, BGM89].

Since Smalltalk-80 has implicit types, it also has implicit subtyping [Ame90]: the implicit types can be related to each other in an implicit subtype (conformance) hierarchy[20]. Although these types are "fictional", the subtype relationships between them follow the same rules as for "normal" object types (Section 2.2.4). In Smalltalk-80, the subtype relationship can basically be reduced to protocol inclusion: a protocol A conforms to another protocol B if A includes all the message selectors of B [Cook92].

In contrast to class-based languages, the implicit subtype hierarchy in Smalltalk-80 is quite distinct from the subclass hierarchy. Instances of a subclass are not generally substitutable for instances of the superclass. The implicit type defined by a subclass is not necessarily a subtype of the implicit type defined by the superclass [BGM89, Ame90]. This is due to the fact that a subclass can substantially modify the behavior of its instances in comparison to the instances of the superclass:

- A subclass may cancel methods it would otherwise inherit from the superclass by reimplementing the methods to generate run-time errors [Cook92]. Consequently, in contrast to the superclass objects, instances of the subclass cannot successfully respond to the corresponding messages any longer. Since type checking is performed at run-time anyway, this approach of deliberately issuing a run-time error is basically equivalent to removing the message from the implicit object type defined by the subclass. In essence, the implicit object type associated with the subclass does not conform to the implicit object type associated with the superclass. The subclass instances respond to the deleted methods with run-time errors.

- A subclass may redefine the behavior of an inherited method. A protocol is a documentation of object behavior, and as such is usually not just a mere set of message selectors, but intuitively associates a certain semantics with each message. For example, the message *less* is typically expected to define an order which obeys certain semantics such as reflexivity, symmetricity, and transitivity. Such nuances can be informally captured by protocols. A subclass may override a method from the superclass with incompatible behavior. Therefore, the subclass instances are not substitutable for instances of the

[20] In fact, although Smalltalk-80 supports only single inheritance, the implicit subtype hierarchy may exhibit relationships where a single type has multiple supertypes [LP91, Cook92].

superclass since their response to certain messages will not comply with the expected semantics.

- A subclass may redefine the signature of an inherited method [BHJL86, BGM89]. As previously determined, formal parameters are constrained by implicit types. Therefore, although a message selector is basically only a name, a signature can be regarded as consisting of more: signatures have implicit function types which are built from the implicit types constraining the formal parameters. Consequently, in order to retain the subtype relationship between the associated object types, a subclass may only change the implicit function type of an inherited method to a subtype of the original signature's type. In principle, a subclass may override any inherited method, arbitrarily changing the associated implicit function types—at the price of losing the substitutability between the superclass and subclass instances. Defiantly retaining this incorrect subtype relationship between such classes will subsequently lead to the occurrence of run-time type errors [Ghe91]. A programmer implementing in a dynamically-typed language must respect the same constraints as would apply to a strongly-typed language, the only difference being the point at which type checking is performed.

It is possible to design types and a subtype hierarchy that are completely unrelated to implementations and the inheritance hierarchy [BGM89, Ame90]. This potential constitutes the flexibility and expressiveness of dynamically-typed programming languages [LP91, Cook92]:

1. Implicit types related by subtyping may be implemented by classes not related by inheritance.

2. The types associated with classes related by inheritance are not necessarily in a subtype relationship.

It proves useful to compute the implicit subtype hierarchy of a class library. The interface hierarchy of the Smalltalk-80 collection classes is computed and analyzed in [Cook92], revealing several instructive cases of conflict between the implementation hierarchy and the subtype relationships.

Chapter 6

Contravariance vs. Covariance

The question of covariant or contravariant specialization of inherited methods has been an issue of ongoing debate in the research community [DT88, DT92, PS92, Sha94, Cas95, BC96, BCC+96, Tho97]. As shown in Section 2.2.3, the substitutability relationship between function types (and consequently object types) is based on contravariant argument types and covariant result types [DT92]. This is typical for strongly-typed languages such as Trellis/Owl [SCB+86] and POOL-I [AL90]. Taking the behavior of the argument types as representative, this is often referred to as *contravariant specialization* [Ghe91, BC96].

However, it has frequently been argued that contravariant specialization is sometimes awkward, counter-intuitive, and the source of many type problems [Har91, DT92, PS92, Cas95], prematurely eliminating various opportunities for code reuse [Sha94]. This chapter will investigate the ramifications of contravariant specialization on object-oriented programming and elucidate an alternative often mentioned in conjunction with statically-unsound type systems [PS92]. By means of *multiple dispatching* this alternative will be type-safely integrated into our present strongly-typed model, avoiding some of the restrictions imposed by the contravariance rule.

6.1 Binary Methods

Contravariant specialization requires that the parameter types of a method are generalized in subtypes, whereas the result type is specialized, ensuring a statically-sound type system [PS92]. However, the practical applicability of weakening the parameter's type demands in subtypes may be seriously questioned[1] [MMM90]. Quite on the contrary, such type constraints imposed to obtain type-safety may be overly stringent for various naturally arising programming situations [PS92]: a programmer will usually want to strengthen rather than weaken a parameter's type demands in a subtype. This controversy can superbly be depicted by one of *the* archetypal examples in common literature [CCHO89, MHH91, DT92, Hür94, BC96, BCC+96]:

Example 6.1: *Point* and *ColorPoint*.
Consider the following implementations:

[1] In effect, many contemporary object-oriented programming languages like C++ [Str97] and Java [GJS96] have confined themselves to *invariant* argument types [Sha94, DE97].

```
implementation Point
    var x : int
    var y : int
    method x() returns int
        return x
    method y() returns int
        return y
    method equal(p : Point) returns bool
        return x = p.x() and y = p.y()
end Point

implementation ColorPoint inherits Point
    var c : string
    method c() returns string
        return c
    method equal(p : ColorPoint) returns bool
        return x = p.x() and y = p.y() and c = p.c()
end ColorPoint
```

Instances of *Point* and *ColorPoint* have the following object types, respectively:

$$\begin{aligned} &\mathit{Point} \equiv \langle\langle x : \mathit{unit} \rightarrow \mathit{real}; y : \mathit{unit} \rightarrow \mathit{real}; \mathit{equal} : \mathit{Point} \rightarrow \mathit{bool}\rangle\rangle \\ &\mathit{ColorPoint} \equiv \langle\langle x : \mathit{unit} \rightarrow \mathit{real}; y : \mathit{unit} \rightarrow \mathit{real}; c : \mathit{unit} \rightarrow \mathit{string}; \\ &\qquad\qquad \mathit{equal} : \mathit{ColorPoint} \rightarrow \mathit{bool}\rangle\rangle \end{aligned}$$

The implementation of *equal* in *ColorPoint* needs to check the argument's color, meaning that the argument type of *equal* must be overridden to *ColorPoint* in the subimplementation. Due to contravariance, the specified function types of *equal* in *Point* and *ColorPoint* would only be in a subtype relationship if *Point* were a subtype of *ColorPoint*. More formally, $\mathit{ColorPoint} \rightarrow \mathit{bool} \leqq \mathit{Point} \rightarrow \mathit{bool}$ requires $\mathit{Point} \leqq \mathit{ColorPoint}$—exactly the opposite of what is intended. The function type of *equal* in *ColorPoint* is not a subtype of the function type of *equal* in *Point*. The former cannot substitute the latter in every context, because it cannot handle at least the same arguments. Therefore, the associated object types are not in the intuitively expected subtype relationship.

These "unfortunate" problems with subtyping, namely that interfaces intuitively appearing to be in a containment relationship are not, result from contravariant specialization combined with recursive types [CCHO89]. Both *Point* and *ColorPoint* are recursive in the argument type of *equal*. Methods whose argument type should be the same as the type of the receiver `self` are not a rarity in object-oriented programming and are usually called *binary methods* [BCC+96]:

Binary Method: A method m of some object x belonging to a type τ is a binary method if m has an argument constrained by type τ. A binary method requires a parameter of the same type as the type of the receiver, namely the type of `self` [Har91, BC96].

The term "binary" traditionally comes from *binary operations*, which take two arguments of the same type and are common in non object-oriented programming languages. Typical examples include binary relations such as $<: int \times int \rightarrow bool$ which take two integer arguments. In object-oriented programming languages, the first argument of the binary operation becomes the message receiver, while the second argument becomes the only method argument. Binary methods are binary in the sense that they act on two objects of the same type: the receiver object and the argument object. In general, a binary method may include other arguments of arbitrary types, with a standard abuse of terminology still referring to them as "binary" methods [BCC+96].

Unfortunately, the type-safe contravariant rule for function subtyping fails with binary methods [CL94a]. The subtyping of binary methods with heterogeneous argument types is not type safe [BCC+96]. Changing the recursive argument type is frequently an implementational necessity [Har91]—the binary method's recursive argument type must change in parallel with the inheritance relationship [BCC+96]. For example, the argument type of *equal* had to be changed covariantly from *Point* to *ColorPoint* in the subimplementation to allow the redefinition of *equal* to query the argument's color.

This could be made more explicit by using the pseudo-type `MyType` to denote the argument's type and the change in subtypes. The argument type changes in parallel, or covariantly, with the alleged subtype relationship [AL90]. Therefore, this behavior is commonly termed *covariant specialization* [Sha94, BC96]. Such covariant specialization of argument types would violate the type-safety guaranteed by contravariance. Consequently, to retain compile-time type-safety, the subtype relationship between the object types associated with the implementations is abandoned.

This loss of subtyping is by no means due to some problem with the definition of subtyping for function types or object types [BCC+96]. Contravariance is not a coincidental result of the combination of some formal rules [Ghe91]. While seemingly counter-intuitive, contravariance is a necessity for a sound type system [CCHO89, Ghe91]. If argument types can be changed covariantly then type-safety is lost, meaning that it becomes possible to write statically well-typed code that generates run-time type errors [Cas95]. To demonstrate how the permission of covariant specialization would inevitably lead to run-time type errors, consider the following example:

Example 6.2: Covariance leads to run-time type errors.
Assume that covariantly specialized argument types are no hindrance for subtype relationships. Then, the objects implemented in Example 6.1 could be reimplemented in a class-based system by exploiting both inheritance and the assumed substitutability of the alleged subtypes.

```
class Point(x : int, y : int)
    var x : int := x
    var y : int := y
    method x() returns int
        return x
    method y() returns int
        return y
    method equal(p : Point) returns bool
        return x = p.x() and y = p.y()
end Point
```

```
class ColorPoint(c : string) subclass of Point
    var c : string := c
    method c() returns string
        return c
    method equal(p : ColorPoint) returns bool
        return super.equal(p) and c = p.c()
end ColorPoint
```

The pseudo-variable `super` is used to bypass the redefinition of a method and to refer to the inherited declaration. The semantics of `super` is treated in [CP89, BC90, Tai96]. We reuse the original method definition in the reimplementation of *equal* in the subclass. The assumed subtype relationship means that objects of *ColorPoint* can be used in any context expecting objects of *Point*. In particular, the overridden methods in the subclass must be able to handle at least the same arguments that can be passed to the original method [BC96]. If the prior class specifications were regarded as legal by the compiler, then the following piece of code would generate a run-time type error[2]:

```
var point : Point := new Point(10, 20)
var colorpoint : ColorPoint := new ColorPoint(10, 20, "blue")
var anotherpoint : Point := colorpoint
point.equal(point)
point.equal(colorpoint)
colorpoint.equal(colorpoint)
anotherpoint.equal(point)
```

The first two comparisons work fine because a point is compared to a point and a color point, respectively. In both cases the *equal* method defined in *Point* is executed, sending the messages x and y to the argument; both points and color points understand these messages. The third comparison executes without error because the argument of type *ColorPoint* is sent the messages x, y, and c; the argument understands these messages because it is guaranteed to be of type *ColorPoint* at compile-time. The expression is type checked using the definition of *equal* in *ColorPoint*, since the receiver is statically known to be at least of type *ColorPoint*. However, due to the covariant redefinition of *equal* in the subclass, the fourth comparison generates a run-time type error:

1. The assignment of *colorpoint* to *anotherpoint* is statically legal, because the source's type is a subtype of the target variable's static type constraint: *ColorPoint* is (assumed to be) a subtype of *Point*.

2. Sending the message *equal* to *anotherpoint* with an argument of type *Point* is statically legal, because *anotherpoint* is known to be at least of type *Point*. The method *equal* defined in *Point* is used at compile-time to type check the expression, confirming that an argument of type *Point* is type-safe. Due to subtype polymorphism, *anotherpoint* may type-safely denote an object of the alleged subtype *ColorPoint*.

[2] Similarly contrived examples leading to run-time type errors can be found in [CCHO89, Ghe91, Hür94, Cas95, BC96, BCC+96].

3. Since the object referenced by *anotherpoint* is actually a *ColorPoint*, and due to the dynamic dispatch of methods, at run-time the message *equal* is associated with the method definition in the receiver's class, namely in *ColorPoint*. In effect, the argument of type *Point* is sent the message *c*—evidently resulting in a type error even though the program successfully passed compile-time type checking.

Objects of *ColorPoint* are not substitutable for objects of *Point*, because the *equal* method of the former cannot substitute the *equal* method of the latter[3]. To obtain type-safety, the signature of *equal* in *ColorPoint* must be forced to have the function type *Point* $\rightarrow$ *bool* [Ghe91, DT92], meaning that the implementation cannot take the argument's color into consideration.

The typing of binary methods is regarded as particularly problematic in the presence of inheritance [BC96, BCC+96]. Section 2.2.4 stated that a type σ is a subtype of another type τ if the messages defined in σ can type-safely replace the messages defined in τ [Car88]. When an inheritance mechanism allows the sharing of code between class specifications, the issue is raised of how the inherited methods may be type-safely overridden, so as to retain the subtype relationship between the associated object types (Section 5.2) [BC96, BCC+96]. The restrictions imposed on method redefinition are governed by contravariant specialization [Ghe91, DT92, Sha94]:

- If a method is inherited without being overridden, then no typing issue is raised, since the inherited method has the same invariant function type in both object types.

- If a method is redefined, then type-safety requires that the new method can replace the old method in every context.

Following the definition of substitutability (Section 2.2.1), this means that the new method's function type must be a subtype of the original method's function type [Ghe91, BC96]. In essence, the presence of binary methods induces two practical problems into object-oriented programming, one concerned with the use of inheritance and the other with the substitutability of object behaviors:

1. In class-based programming languages, the class *ColorPoint* may not be expressed in terms of *Point* because the associated types are not in a subtype relationship. Although the implementations have several issues in common, namely the variable declarations of x, y, and the corresponding accessor methods, these similarities may not be exploited by means of inheritance [Ghe91]. Thus, the intricate typing of binary methods precludes a number of opportunities for code sharing [Sha94, BCC+96]. In a class-based system, the use of `MyType` to denote the type of an argument would mean that the enclosing class cannot have any descendants[4].

2. Since the argument types of binary methods need to be covariantly specialized, the associated object types are not in a subtype relationship. Their respective objects are not substitutable for each other.

[3] The substitutability relationship is valid in contexts that don't send the message *equal* [DT92]. Similar issues can be considered by global program analyses as performed for Eiffel [Jon92].

[4] The use of `MyType` to denote the type of the result would not prevent a subtype relationship.

In confining subclasses to produce subtypes, class-based programming languages must maintain conformance by restricting the programmer's ability to arbitrary change the types of the inherited method's arguments, basically eliminating the use of binary methods [BCC+96]. In other words, if inheritance is restricted to always generate subtypes, then class specifications with binary methods cannot be usefully shared by inheritance [BC96, BCC+96]. Covariant specialization would result in subclasses which no longer produce subtypes [BCC+96], destroying the simplicity of class-based conformance (Section 5.1.2).

This problem can straightforwardly be solved by separating types from implementations and by distinguishing subtyping from inheritance [Ghe91, BCC+96, Roy98]. Inheritance between objects implemented in terms of binary methods can be regained. This approach has already been demonstrated in Example 6.1 and entails the other advantages pertaining to the separation (Section 5.3). The problem for the type system in the presence of inheritance is solved by abandoning the subtype relationship[5].

Unfortunately, the second problem concerning substitutability of behavior is not solved, since the associated object types are still not related by subtyping. Contravariant argument types disallow the useful specialization of binary methods in subtypes [BC96]—colored points are not substitutable for points. However, the client is typically interested in the preservation of this substitutability relationship. The programmer usually wants *ColorPoint* objects to be applicable in any context expecting *Point* objects. Consequently, we are primarily interested in a solution that preserves both compile-time type-safety and the subtyping relations between classes such as *Point* and *ColorPoint* [BCC+96].

6.1.1 Covariant Specialization

In practice, it is frequently necessary to inherit from a superclass and restrict certain methods covariantly [Har91], thus producing descendants which cannot be used as complete substitutes for their parents [Jon92]. Experience shows that covariance of argument types is more common than contravariance [DT88, Ghe91, Cas95].

Due to the limited flexibility of contravariance, some researchers denounce contravariant specialization in favor of covariant specialization, despite its lack of type safety [PS92]. Similarly, the designers of strongly-typed object-oriented languages like Eiffel [Mey97] and O_2 trade type-safety for covariant specialization, putting expressiveness above strict type-safety and attempting to alleviate situations similar to binary methods [CL94a, Cas95, BC96]. This alternative allows the argument types of a message to be specialized in subtypes, meaning that an inherited method may be overridden by a method with subtypes as argument constraints [DT92].

However, as previously demonstrated in Example 6.2, such covariant type systems are not statically sound [CHC90, DT92], because arguments that are seemingly legal at compile-time may in fact have too general types at run-time [PS92]. The type discipline is not safe since type errors may occur even if a program has successfully passed compile-time type checking [BC96].

The problem with covariant specialization is that covariantly overridden methods in subclasses "take away" part of the operation inherited from the superclass [Har91], in the sense

[5] It may be argued whether the problems pertaining to binary methods need to be solved at all. [BCC+96] presents several work-arounds which "ignore" the issues by treating binary methods as normal methods.

that they elide part of the domain handled by the original method. Obviously, these new methods cannot replace the original methods in every context, meaning that the subclass instances are not substitutable for the superclass instances. The interface implemented by the subclass does not contain the interface implemented by the superclass [Har91].

Class-based languages relying on covariant specialization do not abandon the subtype relationship but retain this type-unsafe substitutability property. The identification of types with implementations means that covariantly specialized classes are regarded as subtypes, producing "subtype" instances which are not type-safely substitutable for supertype instances[6] [CCHO89]. Consequently, such type systems usually admit that covariant argument types can lead to run-time type errors and provide different mechanisms to alleviate the problem [DT92]:

- Eiffel [Mey97] claims that descendants generate subtypes, even though inherited methods are covariantly overridden [Cas95, BCC+96]. Most Eiffel compilers compensate for the resulting insecurity in the type system by performing a global, link-time dataflow analysis of the program: the *system validity check* [Jon92, BC96, BCC+96]. This analysis rejects programs that are potentially type-unsafe. Consequently, a program may pass the local, compile-time type check and nevertheless be rejected by the global, link-time analysis [Sha94]. Local type validity does not necessarily imply global type validity [Mey92].

- The compiler automatically inserts run-time type checks to ensure that the actual parameters of a covariantly specialized method are sufficiently specialized [PS92, BC96]. However, such automatic type checks usually only prevent the system from completely crashing, but nevertheless abort with some error message. The net gain of this approach may be questioned, since the difference between an autopilot program "crashing" or aborting with an error message will probably not be significant.

- The responsibility is laid upon the programmer to perform run-time type checks on the actual parameters of a covariantly specialized method [Har91, DT92, Sha94]. This gives the programmer the possibility to dynamically recover from such type errors. For example, the burden of distinguishing between *Point* and *ColorPoint* arguments is placed on the redefinition of *equal* in *ColorPoint*, possibly raising an exception or executing different code if the argument is only a *Point*. However, this approach is equivalent to the practice of analyzing tag fields before dynamic dispatching was introduced by object-oriented programming, entailing the well-known disadvantages in terms of maintainability and modularity [Ing86, Har91].

- The separation of inheritance from subtyping was already discussed above. This approach only solves the problem for the type system in the presence of inheritance, but sacrifices substitutability in favor of covariance [Sha94]. We strive for a solution that preserves a type-safe substitutability relationship.

Covariant specialization in Eiffel and O_2 is gained at the price of local, static type checking and type-safety, respectively. More generally, a type system of an object-oriented programming

[6] The term "subtype" loses the clear meaning in class-based languages like Eiffel and O_2, because the alleged subtype relationship does not completely correlate with the notion of substitutability [BCC+96].

language will necessarily reflect some degree of tradeoff between the following three properties [Tho97]:

1. covariant specialization,

2. complete, local, and compile-time type checking,

3. and subtype polymorphism.

It is impossible for a language to support all three features, because incorporating two of these mechanism in completeness will preclude the integration of the third [Tho97]. For example, reconsider the erroneous code in Example 6.2:

- Covariant specialization and static type checking are reconcilable in the absence of subtype polymorphism, because the language would then be statically-typed and simply disallow the assignment of *colorpoint* to *anotherpoint*.

- Covariant specialization and subtype polymorphism may coexist in the absence of compile-time type checking, leading to the notorious run-time type error.

- Compile-time type checking and subtype polymorphism interact type-safely if the type system is based on contravariant specialization of argument types.

It is important to mention that the issues of covariant and contravariant specialization are not confined to strongly-typed languages. Clearly, they are only issues when subtyping is involved [Har91]. For example, C++ [Str97] does not support function subtyping and thus avoids the entailed problems. Section 5.3.2 showed that languages like Smalltalk-80 [GR83] have implicit subtype hierarchies. Therefore, contravariant specialization is equally present in dynamically-typed object-oriented programming languages. The problems are merely postponed until run-time [Ghe91, Har91]:

- As hinted at the end of Section 5.3.2, the overriding of inherited methods in dynamically-typed object-oriented languages must, in principle, obey the same restrictions as apply to strongly-typed languages. The type constraints of an inherited method's arguments may only be overridden by supertypes.

- The entire responsibility for respecting these constraints is laid upon the programmer—omissions are penalized with run-time type errors since type checking is performed dynamically.

- In contrast, in strongly-typed languages the compiler performs the type checking and reports type errors *before* a program is executed.

Example 6.3: Covariant specialization in dynamically-typed languages.
In a language like Smalltalk-80 [GR83] the implicit argument type of the *equal* method in *ColorPoint* could be covariantly specialized in the descendant. However, the offensive comparison (Example 6.2) would still culminate in a run-time type error [Ghe91].

The principle of substitutability does not allow the relaxation of contravariance without a reconsideration of the run-time method dispatch mechanism [Ghe91]. In the following, we will introduce an intuitively appealing alternative to contravariant specialization, facilitating the employment of covariant specialization without sacrificing type-safety. We solve the problem for both the type system and the client by reclaiming the lost subtype relationship. This approach will be type-safely integrated into our present strongly-typed model.

6.1.2 Towards a Type-Safe Solution

Although it is widely argued that contravariant specialization is counter-intuitive [Har91, DT92, PS92], a practical examination of covariant specialization from a client's point of view reveals it to be just as inappropriate to raise the claim of being the epitome of intuition:

Example 6.4: Covariant specialization is counter-intuitive.
The function type of *equal* in *ColorPoint* (Example 6.2) is $ColorPoint \rightarrow bool$. Essentially, this expresses that instances of *Point* are not legal arguments, and therefore instances of *ColorPoint* are not comparable to instances of *Point*; points and colored points may not interact. However, it is perfectly natural to compare a normal point to a colored point by simply ignoring the latter's color. Thus, covariant specialization, although perhaps necessary from an implementational point of view, reduces the degree of subtype polymorphism available to the client.

We can take a first step towards a type-safe and natural alternative to contravariant specialization by reconsidering the semantics of covariant specialization. Instead of regarding a covariantly redefined method as completely overriding the original method (as in Section 6.1.1), effectively curtailing the domain handled by the original method, we may view covariant specialization as only a *partial* overriding of the previous definition[7] [BC96]. The new definition replaces the original definition in only *some particular cases* [CL94a, Cas95, BC97], namely those explicitly captured by the new method's domain. The new method body is invoked only if the actual run-time arguments comply with the statically specified formal parameter types, otherwise the original method is called.

Contravariance and covariance are not mutually exclusive philosophies [Cas95]: neither contravariant nor covariant specialization can by itself handle all conceivable situations in a satisfactory manner. In other words, any type system that is based on only one of them and excludes the other must handle situations necessitating the other as exceptions [DT92, Cas95]. The need is recognized for a hybrid approach, facilitating both contravariant and covariant redefinition in a strongly-typed, statically type-checked system [Ghe91].

This change of perspective facilitates the reconciliation of contravariant and covariant specialization in a single and sound type system, without having to sacrifice compile-time type-safety [BC96, BC97]. Such a type system would allow defining new languages more expressive than traditional object-oriented programming languages [Ghe91]. Covariance and contravariance appropriately characterize two distinct and independent mechanisms[8] [CL94a, Cas95]:

[7] In contrast, contravariant and invariant overriding still completely shadow the inherited methods.

[8] The independence of the two mechanisms is not a unique characteristic of some peculiar object model but is valid in general [Cas95].

- Contravariance correctly captures the subtyping relationship based on substitutability, establishing whether a set of methods can replace another set of methods in every context [DT92, CL94a, Cas95]. A contravariantly overridden method completely replaces the inherited method definition.

- Covariance characterizes the specialization of code, describing the definition of new code which is intended to replace the old definition in some particular cases [CL94a, Cas95]. The new definition partially overrides the original method only for the arguments it can handle [BC97], while the other, more general arguments are still handled by the old method. Therefore, it cannot happen that the actual parameter to a method has a supertype of the corresponding formal parameter type [BCC+96].

Consequently, the choice *between* contravariance and covariance is a false problem [CL94a, Cas95], because they are not opposing views but quite distinct concepts that both have their irrefutable claim in object-oriented programming. Both should be type-safely incorporated in strongly-typed programming languages.

Example 6.5: Covariance and contravariance.
For example, the method *equal* in *ColorPoint* could be reimplemented by separating the treatment of *Point* and *ColorPoint* arguments into two method definitions. The typing of such multiple method definitions will be elaborated in the next section. Section 6.3.1 will provide the proof that such a definition of class *ColorPoint* is indeed a legal subclass of *Point* because it preserves the subtyping relationship between them [BCC+96].

```
class ColorPoint(c : string) subclass of Point
    var c : string := c
    method c() returns string
        return c
    method equal(p : Point) returns bool
        return x = p.x() and y = p.y()
    method equal(p : ColorPoint) returns bool
        return x = p.x() and y = p.y() and c = p.c()
end ColorPoint
```

1. The first method body specifies the code for arguments of the run-time type *Point*, providing a contravariantly specialized method definition which can substitute the corresponding method of the superclass. This implementation considers only the argument's coordinates.

2. The second method body specifies the code for arguments of the run-time type *ColorPoint*, covariantly specializing the first definition. This implementation considers the argument's coordinates and color.

The programmer can write code for default behavior and specialized code for different argument type combinations [Cha92]. Type-safety stems from the fact that the first method definition ensures complete substitutability towards the superclass, while the second method definition provides specialized code for the particular case in which both the argument and the message receiver are colored points [CL94a].

Due to subtype polymorphism, the decision which method definition is applicable must be postponed until run-time, because the exact type of the actual parameter is not known at compile-time. Similar situations occur whenever the code selection must be based on two or more arguments, rather than on a single receiver [Budd91]. This means that not only the receiver but also the argument types are now involved in the method dispatch process, requiring the introduction of a completely different way of viewing message passing [Ghe91]. A reconsideration of functions and the message passing mechanism follows.

6.2 Multiple Dispatching

The alternative interpretation of covariant specialization necessitates a method dispatch mechanism that takes into consideration the run-time types of the actual arguments. *Multiple dispatching* differentiates itself from conventional *single dispatching* in that the latter uses a single, privileged argument (the message receiver) to determine the method to be executed, while multiple dispatching can base the lookup on more than a single argument [LW90, BCC+96]. In Example 6.5, the method body executed depends on the run-time type of the receiver *and* the argument passed to *equal*; the colors are considered in the comparison only if both objects are of type *ColorPoint*. *Multiple dispatching* will allow us to model both traditional and more expressive forms of message passing [Ghe91].

6.2.1 Overloaded Functions

Prior to pursuing the issue of multiple dispatching for object-oriented programming languages, we shall consider the more generally applicable case for traditional programming languages based on functions. For this reason we introduce the notion of *overloaded functions*[9] [Cas95, BCC+96]:

Overloaded Function: An *overloaded function* is a collection of several function bodies associated with a single function name. The overloaded function consists of a set of ordinary functions, called *branches* of the overloaded function[10].

An ordinary function can be considered as a special case of an overloaded function with only one branch [Cas95]. An overloaded function is built by starting from a regular function (considered as a single-branched overloaded function) and successively adding new branches [Cas95, Ghe91]. Since we want overloaded functions to be at least as expressive as ordinary functions, in particular we want to define a substitutability relationship for them as well as attach them to objects in an object-oriented style, overloaded functions must be typed [Ghe91, Cas95, BCC+96]:

[9] The term *multi-method* is frequently used synonymously with overloaded functions [Cha92, BCC+96]. However, "multi-method" suggests some notion of encapsulation as in single-dispatching languages, which, in general, is not the case for overloaded functions. We confine the term to the context of *encapsulated multi-methods* which are treated in Section 6.3. Overloading as in C++ is only a compile-time concept, while the notion of overloaded functions dealt with now is resolved dynamically. More on this in Section 6.2.2.

[10] Refer to [Ghe91, Cas95, CGL95] for a more theoretical study of overloaded functions, in particular an extension of the simply-typed λ-calculus, the $\lambda\&$-calculus, encompassing overloaded functions, subtyping of overloaded function types, and branch selection.

Overloaded Function Type: An overloaded function type F, constructed from the set of ordinary function types $S_1 \to T_1, \ldots, S_n \to T_n$, is written as $\{S_1 \to T_1, \ldots, S_n \to T_n\}$[11]. The type F describes the overloaded functions that satisfy the associated predicate ψ_F, defined as follows: an overloaded function f has the type $\{S_1 \to T_1, \ldots, S_n \to T_n\}$, meaning that it is an element of F ($\Leftrightarrow \psi_F(f)$), if for every function type $S_i \to T_i$ in F there is a branch in f that has that function type. By definition of function subtyping (Section 2.2.3), the corresponding branch in f may actually have a subtype of the required type $S_i \to T_i$[12]. The predicate ψ_F can informally be expressed as: $\psi_{S_1 \to T_1, \ldots, S_n \to T_n}(f) = \forall i \in \{1, \ldots, n\} : f$ *has a branch with at least type* $S_i \to T_i$.

Example 6.6: Overloaded functions.
Consider the following overloaded function definition:

> `function` $equal(p : Point,\ q : Point)$ `returns` $bool$
>   `return` $x(p) = x(q)$ `and` $y(p) = y(q)$
> `function` $equal(p : ColorPoint,\ q : ColorPoint)$ `returns` $bool$
>   `return` $x(p) = x(q)$ `and` $y(p) = y(q)$ `and` $c(p) = c(q)$

The function *equal* is defined as a global function, fostering a programming style similar to functional programming languages, rather than being defined as a method in the usual object-oriented style[13]. The function *equal* consists of two branches: one for arguments of type *Point* and another for arguments of type *ColorPoint*. The second branch is called only if both arguments are of run-time type *ColorPoint*. If any one of the arguments is of run-time type *Point* then the first branch is executed. The function has the overloaded function type $\{Point \times Point \to bool, ColorPoint \times ColorPoint \to bool\}$.

Example 6.7: Mixed comparisons.
Overloaded functions allow the specialization in different ways for each possible combination of actual arguments [BCC+96]. For example, the overloaded function from Example 6.6 could be augmented by adding branches for mixed comparisons [Ghe91, MHH91]:

> `function` $equal(p : Point,\ q : ColorPoint)$ `returns` $bool$
>   `return false`
> `function` $equal(p : ColorPoint,\ q : Point)$ `returns` $bool$
>   `return false`

The function *equal* now has the overloaded function type
$\{Point \times Point \to bool, ColorPoint \times ColorPoint \to bool,$
$Point \times ColorPoint \to bool, ColorPoint \times Point \to bool\}$.

The inherent type of an overloaded function consists of the set of function types of its branches [Cas95, BC96, BCC+96]. An ordinary function type (Section 2.2.3) can be considered as an overloaded function type consisting of a single function type [Cas95]. Consequently, the

[11] Naturally, an overloaded function may take multiple arguments [BCC+96]. The respective S_i may be constructed types such as function types $U \to V$ or cartesian product types $U_1 \times \cdots \times U_n$.

[12] Therefore, the presence of several function types of F may be satisfied by a single branch in f.

[13] Similarly, the methods x, y, and z are replaced by global functions that extract the appropriate information from their arguments.

inherent overloaded function type of an ordinary function is the singleton set containing its function type [BC96, BCC+96].

Similarly to the definition of function types and function subtypes in Section 2.2.3, an overloaded function f belongs to an overloaded function type F if the inherent type of f is a subtype of F:

Subtyping Overloaded Function Types: An overloaded function type F_1 of the form $\{S_1 \to T_1, \ldots, S_m \to T_m\}$ is a subtype of another overloaded function type F_2 of the form $\{U_1 \to V_1, \ldots, U_n \to V_n\}$ if every overloaded function of type F_1 also has type F_2. In terms of predicates, we require $\psi_{\{S_1 \to T_1, \ldots, S_m \to T_m\}}(f) \Rightarrow \psi_{\{U_1 \to V_1, \ldots, U_n \to V_n\}}(f)$ for all overloaded functions f. Substitution leads to ($\forall i \in \{1, \ldots, m\} : f$ *has branch with* $S_i \to T_i$) $\Rightarrow$ ($\forall j \in \{1, \ldots, n\} : f$ *has branch with* $U_j \to V_j$). In other words:

- The overloaded function type $\{S_1 \to T_1, \ldots, S_m \to T_m\}$ is a subtype of $\{U_1 \to V_1, \ldots, U_n \to V_n\}$ if for every $U_j \to V_j$ in the latter there exists a $S_i \to T_i$ in the former with $S_i \to T_i \leqq U_j \to V_j$.
- Several branches of the supertype may be replaced by a single branch in the subtype [Cas95], meaning that the subtype may actually have less branches than the supertype.
- Similarly to object subtypes having more messages than their supertypes (Section 2.2.4), the overloaded function subtype may have additional branches which are unrelated to the supertype's branches, defined on completely different and unrelated domains.

The subtyping rules for overloaded function types can be intuitively explained in terms of substitutability. The overloaded function type F_1 is a subtype of another overloaded function type F_2 if expressions of the subtype F_1 can be used in any context expecting expressions of the supertype F_2 without causing run-time type errors [Cas95]. This is exactly the case if, for every function type contained in F_2, there exists a function type in F_1 that is a subtype and can therefore substitute it [BCC+96]. In other words, an overloaded function can replace another overloaded function (of different type) if, for every branch that can be selected in the latter, there is at least one branch in the former that can replace it (using the conventional rules for function subtyping) [Cas95, BC96, BCC+96]. See Section 6.3.1 for some examples.

6.2.2 Single vs. Multiple Dispatching

As already mentioned, the compile-time type information about the argument expressions may not be sufficient to select an appropriate function branch statically [MHH91]. Therefore, the decision which overloaded function branch is applicable must be postponed until run-time, since the types of the actual parameters are typically not known until then. In general, it is impossible to choose the branch to be executed according to the type of a single argument—the selection of the overloaded function branch requires knowledge about the dynamic types of all arguments [BCC+96].

The code executed depends on the run-time types of the arguments, meaning that the traditional function application and message passing mechanisms must be reconsidered [Ghe91, DT92]. In particular, the dispatching of overloaded functions is substantially different from binding ordinary functions or dispatching methods in conventional object-oriented programming languages [BKK+86, Ghe91, MHH91, Cha92, BP93, CL94b, Cas95, BCC+96, BC97]:

- Ordinary functions as in C++ [Str97], Ada [Ada83], and Lisp are bound at compile-time, based on the static types of the arguments.

- The usual style of dispatching methods in object-oriented programming languages such as C++ [Str97] and Smalltalk-80 [GR83] is based on the *implementation* of a distinguished and privileged object—the message receiver. The dynamic types of the normal arguments are not considered in the dispatch and do not affect which code is executed. This mechanism is called *single dispatching*, because the code selection is based on a single, mostly implicitly passed argument called `self` or `this`. The implementation of the receiver alone drives the code selection[14].

- Multiple dispatching as in Cecil [Cha92], CLOS [Kee89], and CommonLoops [BKK+86] is a more general form of message passing. The selection of the overloaded function branch to be executed is performed at run-time, based on the dynamic types of all[15] function arguments rather than just a single receiver. In general, due to subtype polymorphism, the run-time types of the actual arguments will only be subtypes of the statically declared types of the corresponding expressions.

Resolving a function application or method call on the basis of the run-time types of the arguments is called *late binding* or *dynamic dispatching* [Cas95], while resolving the call based only on the compile-time types is called *early binding* [Ghe91]. Evidently, this distinction only affects languages with some form of subtype polymorphism; otherwise the run-time type of an expression *is* its compile-time type and the language can be statically-typed [Ghe91]. In fact, dynamic dispatching is a necessary consequence of subtype polymorphism, ensuring that the most specific version of a method is applied in response to a message send [Cas95].

Note that there is a considerable difference between our notion of overloaded functions and the concept of overloading in languages like C++ [Str97] and Ada [Ada83]. C++ offers "overloaded functions" but, in contrast to our notion of overloaded function, that in C++ is based on early binding:

- The overloading featured by C++ is resolved statically [Har91, BC96], meaning that the selection of the function code is performed at compile-time using the declared, static types of the arguments rather than their run-time types [Budd91, Cha92, Str97]. The dynamic types of arguments do not affect which code is executed: at run-time the code for such functions has already been selected [Cas95, BCC+96].

[14] Another problem with methods is their asymmetrical nature: arithmetic expressions such as `3+4` must be coded as `3.plus(4)` in purely object-oriented languages, giving one of the objects an awkwardly privileged role [Cha92, BCC+96]. This problem can be alleviated by translating the method into an overloaded function as in Example 6.6, thus dispatching on both arguments.

[15] In Cecil [Cha92] the arguments to drive the run-time branch selection can be denoted individually. Only these *specialized* arguments are considered in the dispatch process and the others are not. This allows the integration of all three dispatch models into a uniform framework, depending on whether a function is specialized on zero, one, or multiple arguments, respectively [Cha92, CL94a]. We do not require this distinction because in [Cha92] it is introduced mainly to alleviate the problems of encapsulation and multiple dispatching: the function has privileged access *only* to the arguments it explicitly dispatches on. We solve these problems by merging overloaded functions with method encapsulation in Section 6.3.

- On the contrary, our notion of overloaded functions determines the function body on the basis of the dynamic types of the arguments [BCC+96].

6.2.3 Static Type-Checking and Dynamic Branch Selection

Unfortunately, a wider acceptance of overloaded functions is partially hampered by the absence of compile-time type checking in most existing languages supporting multiple dispatching [CL94b]—the majority of programming languages that support overloaded functions are dynamically-typed. The language Cecil [Cha92] is one of few exceptions in being a strongly-typed language that provides a mechanism for overloaded functions [BCC+96]. In this and the next section we elaborate the issue of how overloaded functions can be type-safely integrated into strongly-typed, statically type-checked programming languages.

Although the overloaded function branch that is ultimately executed is typically determined at run-time[16], it is desirable to statically check the use of overloaded functions for type correctness [MHH91]. In particular, a type-checking algorithm is required that certifies the consistency of a group of overloaded function branches and statically guarantees the absence of branch selection errors [ADL91, CL94b].

As with conventional single dispatching languages, overloaded function call sites are statically type-checked by using the compile-time types of the arguments to compute the compile-time type of the result [LW90, BC97]. Overloaded functions obey a two-level selection discipline:

1. At compile-time, a standard overloading resolution mechanism as in C++ [Str97] or Java [GJS96] is performed for every overloaded function invocation, choosing a "best matching" branch based on the compile-time types of the arguments [Ghe91, BC97]. This branch is employed to determine the compile-time type of the result. To retain type-safety, the dynamic type of the actual result may only be a subtype of the statically computed compile-time type.

2. At run-time, the "best matching" branch to be executed is determined on the basis of the run-time types of the actual parameters[17] [Ghe91, Cas95, BCC+96, BC96]. Therefore, the branch executed in response to an overloaded function application depends on the dynamic types of all arguments [BP93].

If the type of the argument is exactly one of the formal parameter types of the overloaded function branches, then the corresponding branch is selected. However, due to subtype polymorphism, when an overloaded function of type $\{S_i \to T_i\}_{i \in \{1,\dots,n\}}$ is applied to an argument x, the compile-time or run-time type of x may not match any of the branch argument types S_i, but just be a subtype of one of them [Ghe91, Cas95]. Therefore, the term "best matching" is used to denote that in both cases the branch is chosen whose argument type S_i best approximates (in terms of subtyping) the type of x [Cas95]. The most specifically applicable

[16] The language Kea [MHH91] supports *multivariant functions*, a variation of overloaded functions and similar to the *encapsulated multi-methods* presented later. [MHH91] show that the dynamic dispatch can, in some cases, be prematurely resolved at compile-time. Nevertheless, in general, the branch selection must be performed at run-time.

[17] See [Cha92] for a concrete approach to determining the most specific branch by ordering the potential branches according to their type constraints [ADL91, CL94b].

branch is chosen [BKK+86], meaning that the system selects the branch defined for those formal parameters whose types best match the static (in the first case) or dynamic (in the second case) types of the arguments[18] [BCC+96]:

Branch Selection: If the type of an overloaded function is $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ and this overloaded function is applied to an argument with compile-time or run-time type S (depending on whether the branch selection is used to determine the result type at compile-time or the branch to be executed at run-time), then the best matching branch j is selected such that $S_j = \min_{1,\ldots,n} \{S_i | S \leqq S_i\}$ [Ghe91, Cas95, BC96, BCC+96]. In any case, to ensure type-safety, the selected S_j must be supertype of the argument type S [Ghe91].

A branch is only applicable to actual arguments whose types are subtypes of the branch's formal parameter types. Consequently, for every overloaded function application there must exist a branch with formal parameter types at least as general as the compile-time types of the actual arguments. Thus, the type-checker can guarantee that at least one branch will be applicable at run-time [Har91]. Without compile-time type checking it would be possible that not a single branch can be selected because the formal parameter types of all branches are (at least in part) more specific than the run-time types of the arguments [Har91, MHH91].

Another situation that must be considered and avoided by the compiler is the *ambiguity* of function calls—the algorithm to determine the best matching branch may have to decide between several, equally specific branches at run-time [ADL91, MHH91, Cha92, BC97, CC99]. This situation is solved in [BC97] by giving the branches different priorities according to the textual order of their definitions in the source code. In case of an ambiguity, the textually last branch is selected.

6.2.4 Well-Formedness

As elaborated in the previous section, every overloaded function application must be associated with a result type: static type-checking requires that this type is computed at compile-time, although the overloaded function branch to be executed is usually selected at run-time. However, the branches of an overloaded function may have both related and unrelated domains and codomains, respectively. Consequently, how can the result type of the function application be type-safely computed at compile-time? The following example will make clear that, in essence, this raises the question of which sets of branches can be considered to form type-safe overloaded functions—does a strongly-typed language allow any arbitrary set of function types to form a well-typed overloaded function type [Ghe91, Cas95, CGL95, BCC+96]?

Example 6.8: Type-unsafe overloaded function.
Assuming that the type *ColorPoint* is a subtype of *Point*, consider the following definition of an overloaded function with two branches:

[18] Single dispatching in a strongly-typed language typically requires a single table lookup to find the appropriate method body. In contrast, selecting the best matching branch based on the run-time types of all arguments may be a comparatively expensive operation [CC99]. [Ing86, BC96, BC97] implement multiple-dispatching as a chain of single dispatches, meaning that programs using only single dispatching in such languages are not penalized [BCC+96].

> `function` $break(p : Point)$ `returns` $ColorPoint$
>   `return new` $ColorPoint$
> `function` $break(p : ColorPoint)$ `returns` $Point$
>   `return new` $Point$

The following piece of code would lead to a run-time type error:

> `var` $p : Point :=$ `new` $ColorPoint$
> `var` $cp : ColorPoint := break(p)$
> $cp.c()$

The variable p is statically declared to denote objects of type *Point*. Due to subtype polymorphism, the assignment of a *ColorPoint* object to p is legal. Because the compile-time type of p is *Point*, the first branch of *break* must be used to compute the compile-time result type of the function application—an object of *ColorPoint*. Therefore, the assignment to cp (constrained by *ColorPoint*) is legal. No problem would arise if the run-time argument to *break* really was of type *Point*. Unfortunately, p actually denotes a *ColorPoint* object, meaning that at run-time the second branch is selected and returns an object of type *Point*. Consequently, the variable cp now illegally references a *Point* object, leading to a run-time type error in the last statement.

Not every set of function types forms a well-typed overloaded function type [Cas95]. To have a strong and sound type system, it is necessary to preserve the property that the compile-time type of any expression is always a supertype of the possible run-time type of the expression value [Ghe91]. A rule is required to guarantee that during computation the run-time type of an expression can never become more general than its compile-time type. The type of a term may only become more specific during computation [ADL91, Cas95, BCC+96, BC97]:

Well-Formedness: A set of function types $\{S_1 \to T_1, \ldots, S_n \to T_n\}$ produces a well-formed overloaded function type if for all $i, j \in \{1, \ldots, n\}$: if $S_i \leqq S_j$ then $T_i \leqq T_j$. This condition is also known as the *covariance condition*[19] [Cas95, CGL95].

In other words, although the different branches that compose an overloaded function are not necessarily required to return the same result type, for type-safety it suffices that the covariance condition is fulfilled [BCC+96]. Well-formed overloaded function types are those types where all function type components are mutually consistent [Ghe91, BC97], meaning that if the argument types are in a subtyping relationship, then the result types must be related in the same way [Ghe91, BCC+96]. For each pair of overloaded function branches $S_i \to T_i$ and $S_j \to T_j$ with the same name and number of arguments[20]: if the argument type S_i is a subtype of S_j then the corresponding result type T_i must be subtype related to T_j [LW90, BCC+96].

In general, the compile-time type of an expression is only a supertype of the run-time type of the corresponding value. Consequently, as shown in Example 6.8, the overloaded function branch used to type-check a function application may not be the same as the branch

[19]Some further conditions are required to assure the existence and uniqueness of a best matching branch for a given function application [BP93, Cas95, CGL95, BCC+96].

[20]The branches of an overloaded function must not necessarily all have the same arity. Those branches with the inappropriate arity are simply not considered at function application [BCC+96].

actually selected at run-time based on the dynamic argument types [Ghe91]. The covariance condition assures that the compile-time type of the function application is always a supertype of the run-time result type, regardless of the actual run-time argument types and dynamically selected branch[LW90, Cas95].

To illustrate the indispensability of the covariance condition with a more formal example [Ghe91, Cas95], assume that the type S_1 is a true subtype of S_2 ($S_1 \leqq S_2$ and $S_1 \neq S_2$) and that S_1 and S_2 are the respective argument types of the branches of an overloaded function f of type $\{S_1 \to T_1, S_2 \to T_2\}$. If f is applied to an argument x of compile-time type S_2, then the first branch is used to statically type-check the function application. Thus, the inferred compile-time type of the result will be T_2, meaning that the dynamic type of the actual result may only be T_2 or a subtype thereof. Depending on the run-time type of x the following situations can occur:

- If the run-time type of x is S_2, then the first branch with type $S_2 \to T_2$ is dynamically selected and the result type is T_2, which is clearly in compliance with the expected result type.

- However, if the run-time type of x is S_1, as possible due to subtype polymorphism, then the second branch with type $S_1 \to T_1$ is dynamically selected and the result type is T_1. If T_1 is not a subtype of T_2 then the actual result type is not compatible with the expected result type: a run-time type error is inevitable.

Consequently, to assure that the run-time type of the result is always a subtype of the compile-time of the result of an overloaded function application, the type T_1 must be a subtype of T_2 in correspondence with the associated argument types S_1 and S_2 [Cas95, Ghe91, BCC+96]. If the argument types S_1 and S_2 are not related by subtyping, then the restrictions on the result types become obsolete, because the selection of one of the branches for compile-time type checking inherently excludes the other from being selected at run-time [Cas95].

6.2.5 Overloaded Functions vs. Encapsulation

Although multiple dispatching is more expressive than single dispatching, it is supported by only a few object-oriented programming languages [Cha92]. A wider acceptance of multiple dispatching is hampered by several drawbacks [CL94b], including the increased language complexity [Cha92] and the more complicated semantics of method lookup. This section will review the difficulties encountered with multiple dispatching, paving the way for the reconciliation of overloaded functions with object-oriented programming in section 6.3.

The main problem pertaining to overloaded functions in the context of object-oriented programming is that the increased expressiveness of multiple dispatching is usually obtained at the expense of method encapsulation [CL94b, BCC+96]. Overloaded functions as in [ADL91, Cha92] do not preserve the notion of encapsulation because there is no privileged receiver to which a message is sent [BC96]. Overloaded functions are defined globally and separately from object specifications; objects only "encapsulate" data in terms of instance variables. Consequently, the issue of encapsulation is frequently ignored in object-oriented languages like CLOS [Kee89] and Flavors [Moon86], which induce a different style of programming from traditional single-dispatching object-oriented programming languages [Har91, BCC+96].

Overloaded functions are typically considered to be "outside" of the objects they manipulate, in contrast to methods which internalize the "object-oriented feeling" of sending messages to a distinguished receiver [MHH91, Cha92, Cas95, Har91].

In traditional, single-dispatching object-oriented languages a program is usually organized around abstract data types in terms of objects with encapsulated state and operations—the instance variables and the associated methods. Computation is performed by sending messages to receivers. In contrast, the shift to overloaded functions has the following implications:

- By lack of linguistic support, a majority of the existing multiple-dispatching languages does not encourage the data-abstraction-oriented programming style of conventional object-oriented languages [Cha92]. Instead, they tend to foster a programming style organized around overloaded functions [Har91, Cha92, BCC+96], which are not defined as part of any single object implementation but externally to all objects [Höl93]. Consequently, the classical object-oriented paradigm of sending a message to a receiver must be radically reconsidered in such multiple-dispatching languages. In this sense, programming with overloaded functions has more similarities to function-oriented programming and "does not feel object-oriented" [Cha92, CL94b].

- Single-dispatching languages enforce object-level encapsulation by permitting only an object's methods to directly access that object's instance variables, meaning that the manipulation of an object's state is localized to the object's methods. In multiple-dispatching languages, the overloaded functions do not belong to one particular object implementation but are treated as being outside of all objects they manipulate. This view of overloaded functions precludes the encapsulation of the internal implementation details of a single object. Overloaded functions require privileged access to all their arguments [Cha92].

- The definition of branch selection in Section 6.2.3 stated that the best matching branch is determined on the basis the argument types. However, since overloaded functions necessitate privileged access to the instance variables of the arguments they manipulate, the code of the overloaded function branches must rely on the arguments having specific implementations in order to allow access to the argument structures. Consequently, the dispatching mechanism must use particular object implementations rather than object types to drive the branch selection [Cha92, BCC+96]. This means that the formal parameters of an overloaded function are not constrained by types any longer but by concrete implementations. The actual parameters must inherit from these argument constraints rather than being subtypes[21] [Cha92].

In summary, there no longer exists the notion of a privileged, distinguished receiver of a message because an overloaded function is applied to several arguments which equally participate in the run-time branch selection. Objects do not encapsulate methods any more but represent just data; a class of objects is characterized by the internal variables of its instances. The application of an overloaded function to a set of arguments is equivalent to a global function which is dynamically bound to different function bodies depending on the internal implementations of the actual arguments at run-time [BCC+96].

[21] Therefore, although Cecil [Cha92] has separate notions for describing types, implementations, subtyping, and inheritance, multiple dispatching is nevertheless based on implementations and inheritance.

The language Cecil [Cha92] tries to solve these encapsulation problems by changing the view of overloaded functions. The programmer can individually denote which arguments should be *specialized* to drive the branch selection at run-time. The overloaded function is viewed as "inside" these specialized arguments and therefore has privileged access to them, but not to arguments that do not drive the branch selection. Arguments that participate in the branch selection are constrained by concrete implementations, while arguments that do not participate in the branch selection may be constrained by particular types (these constraints can be checked at compile-time [CL94b]). A further possibility is to introduce a separate module system to control the direct access to instance variables by textual restriction [Cha92, CL94b]. While these proposals are only partial alleviations to the enumerated problems, the following section will present a more complete and effective solution.

Finally, note that different languages have different notions of encapsulation. The encapsulation concept in C++ [Str97] bears subtle differences to encapsulation as encountered in Smalltalk-80 [GR83] and *Susi* [Kur99]:

- Smalltalk-80 and *Susi* support "object-level" encapsulation. Only an object may access its own instance variables. In particular, no object may access the internal representation of another object, regardless of whether the two objects are instances of the same class or not.

- C++ supports "class-level" encapsulation, fostering a style derived from languages based on abstract data types such as CLU [LSAS77]. An object has privileged access to all elements of its type. Since types are identified with classes, this is equivalent to saying that an object has privileged access to all instances of its class [Liu91, Str97].

6.3 Encapsulated Multi-Methods

In this section we will merge the concept of overloaded functions with the notion of encapsulation which is immanent to object-oriented programming, striving to solve the problems described in the previous section, typically adhering to overloaded functions and encapsulation. The presented approach will facilitate the retention of multiple dispatching even in the presence of privileged message receivers [BC96].

Object-oriented programming is typically based on a model similar to Smalltalk-80 [GR83], where a function is not defined globally, but as a method of some object. Every method has a privileged receiver argument denoted by `self`, which is the only argument whose instance variables can be directly accessed by the method [BCC+96]. The transition from function-oriented programming to object-oriented programming was undertaken over three decades ago: normal functions were embedded into objects and granted privileged access to their object's state.

The reconciliation of overloaded functions with encapsulation can be performed analogously. Instead of defining overloaded functions globally, they define the method bodies in an object implementation [BCC+96]. In other terms, overloaded functions are embedded into objects and granted privileged access to their object's instance variables. This leads to the notion of *encapsulated multi-methods*[22] [BC96, BCC+96, BC97]:

[22] [BC96] describes how encapsulated multi-methods can be retrofitted into existing covariant programming

Encapsulated Multi-Method: An encapsulated multi-method is an overloaded function that is encapsulated in an object. The type of an encapsulated multi-method is an overloaded function type. An encapsulated multi-method may have several branches which correspond to normal methods. The branch to be executed in response to a message send is determined at run-time based on the dynamic types of the message arguments.

Apart from that, encapsulated multi-methods are like normal methods in the usual, object-oriented sense, meaning that they share the common properties of conventional methods[23]:

- A multi-method branch is invoked in response to a message send [BCC+96].
- A multi-method has direct access to its object's instance variables. In particular, a multi-method may not access the instance variables of its arguments [MHH91].
- The implementation of a multi-method branch may refer to the receiver of the current message via `self` [BCC+96].

Multi-methods are associated with messages just like normal methods. The dispatch of encapsulated multi-methods is a combination of the traditional method lookup in single-dispatching languages, based on the *implementation* of the message receiver, and the branch selection algorithm of multiple-dispatching languages, based on the run-time *types* of the message arguments. Effectively, the branch to be invoked at run-time in response to a message send is determined in two steps [BCC+96]:

1. When a message is sent to a receiver, in accordance with the conventional lookup in object-oriented programming, the receiver's implementation must provide a method that can be associated with the sent message. This method may be a conventional method or an encapsulated multi-method.
2. If the selected method is a normal method then it is invoked. If the selected method happens to be a multi-method, then, as elaborated in Section 6.2.3 for overloaded functions, the best matching branch is determined on the basis of the run-time types of all further parameters (without the receiver). The selected branch is finally executed using the standard semantics of method application.

The first phase determines the method based solely on the receiver's implementation and ignores the other arguments, thus retaining the antisymmetricity of conventional single-dispatching languages. However, if the selected method is a multi-method, then the second phase uses the run-time types of all arguments to select the best matching branch from those defined for that encapsulated multi-method. In other words, single dispatching selects a method and multiple dispatching selects a branch.

languages such as O_2, thus retrospectively ensuring the type safety of code that uses covariant specialization. [BC97] proposes the addition of *parasitic methods* (a variation of encapsulated multi-methods) to the Java programming language.

[23]In the following, we will use the term "multi-method" synonymously with "encapsulated multi-method", quite contrary to [Cha92, BCC+96] who use the term "multi-method" to denote overloaded functions.

The definition of subtyping for object types from Section 2.2.4 can be extended to accommodate multi-methods. The subtyping rules for overloaded function types (Section 6.2.1) can be employed to establish subtyping relationships between objects with multi-methods. The multi-methods of the subtype must be able to substitute the corresponding multi-methods of the supertype in any context. By regarding normal methods as one-branched multi-methods [BC96], it is even possible to establish subtyping between objects implemented with multi-methods and objects implemented without multi-methods.

In reminiscence of Section 6.2.5 we can see that encapsulated multi-methods solve the problems of encapsulation while largely preserve the advantages and expressiveness of multiple dispatching [BCC+96]:

- Messages are sent to distinguished receivers, meaning that the well-established message passing paradigm and "object-oriented feeling" can be retained, albeit with a slight modification of semantics (expressed by the second dispatch phase described above).
- Multi-methods are granted privileged access only to the instance variables of the object they belong to, namely the message receiver. The manipulation of an object's internal state is localized to the object's (multi-)methods, retaining the conventional notion of encapsulation.
- Only the first dispatch phase is based on the implementation of the receiver, since the (multi-)method requires direct access to the receiver's state. However, since the multi-method does not need to obtain privileged access to the further arguments, the branch selection can be based on the argument types rather than on their implementations[24] [BCC+96].

In accordance with Chapter 4, multi-methods can be inherited and overridden just like normal methods. For the sake of simplicity, we assume the redefinition of a multi-method completely overrides the inherited multi-method, meaning that all the branches of the inherited multi-method are shadowed[25] [BC96, BCC+96]. It is not possible to inherit some branches and override others. If a single branch is overridden, then all the remaining branches must be specified from scratch. The following section will provide some concrete examples involving encapsulated multi-methods.

6.3.1 Type-Safe Binary Methods

By virtue of encapsulated multi-methods, we can accomplish the type-safe integration of covariant specialization in strongly-typed object-oriented programming [Cas95, BCC+96]. Encapsulated multi-methods allow the benefits of inheritance and covariant specialization to be retained without violating the subtype relationship based on contravariance [MHH91, BC96]. This section will demonstrate that the intuitively motivated change of perspective proposed in Section 6.1.2 has a natural interpretation and implementation in terms of encapsulated multi-methods, permitting a natural and type-safe definition of binary methods in object-oriented programming [Har91, BC97].

[24] In class-based systems the dispatch is on classes.

[25] Other models are possible in which inheritance and overriding of multi-methods are on a per-branch basis [Ghe91, Cas95].

The fundamental problem with a covariantly specialized method is that it cannot completely override the inherited method because the principle of substitutability requires that the new definition must be able to handle at least the same arguments as the definition to be replaced. This deficiency can be compensated by providing additional code defined on the original domain, specifying the actions to be taken in case the actual arguments are not specific enough for the covariantly specialized method [BCC+96]. In other words, if one wishes to specialize a method with covariant parameter types, it is nevertheless necessary to add code that can handle the arguments that could be passed to the method being replaced [BC96, BG96].

In this sense, we regard a covariantly specialized method as only a *partial* redefinition of the original method [MHH91]. This new definition substitutes the original definition in only *some particular cases* [CL94a, Cas95, BC96, BC97], namely in those cases in which the run-time method arguments can be handled by the covariantly specialized definition. The more general arguments are still handled by the old definition. Figuratively speaking, the original definition and the covariantly specialized definition cooperate in a joint venture to provide the new functionality. The original definition retires some of its initial competence to the new definition: if the run-time argument types fall into the domain of the covariantly specialized definition, then the original definition is ignored in favor of the more specialized definition.

This interpretation of covariant specialization can be directly mapped to encapsulated multi-methods by viewing the original definition and the covariantly specialized redefinitions as distinct branches of a single multi-method contained in the object. A subset of the branches establishes the substitutability relationship towards the supertype's method, while the remaining, covariantly specialized branches represent more specific code for covariantly specialized argument types [MHH91]. Because the definition of the covariantly specialized branch overrides all the branches of the inherited multi-method, the original branches must be respecified in the derived implementation. The applicability of the covariantly specialized branches to the actual arguments is determined dynamically [Ghe91].

Following this point of view, contravariant and covariant specialization are neither mutually exclusive nor opposing mechanisms but both represent distinct and independent concepts in object-oriented programming with encapsulated multi-methods [Ghe91, Cas95]:

- Contravariance is necessary to establish the subtyping relationship between overloaded function types. A multi-method can only substitute another multi-method of different type in any context if, for every branch in the latter, there exists at least one branch in the former that can replace it. To preserve the subtype relationship between object types, a subset of a multi-method's branches in the subtype must establish the substitutability relationship to the corresponding multi-method in the supertype. Since branches have normal function types, substitutability between branches is based on contravariant argument types (Section 2.2.3).

- Covariance can be used to describe the specialization of code. Covariantly specialized branches partially override the other multi-method branches by replacing them if they can handle the run-time arguments. The more general arguments are handled by the more general branches.

Encapsulated multi-methods allow binary methods to be type-safely overridden in class-based programming languages. When a binary method is overridden in the subimplementation, it is not sufficient to define the binary method for arguments of the subtype. To retain substitutability, it is also necessary to define how the binary method should handle arguments of the supertype [BCC+96]. In other words, when defining a binary method for the types A and B, and $A \leqq B$ is required, then implementations of A must specify not only the binary method's behavior for arguments of the subtype A, but also the behavior for arguments of the supertype B [Cas95].

Example 6.9: *Point* and *ColorPoint* with encapsulated multi-methods.
As already pointed out in Section 6.1.2, the problem with the definition of *ColorPoint*'s method *equal* in Example 6.2 is that it cannot handle arguments of type *Point* [BCC+96]. The substitutability relationship can be regained by providing an additional method body for *equal* defined on objects of the supertype *Point*. These definitions in *ColorPoint* are considered to constitute a single, encapsulated multi-method with two branches [BC96, BCC+96]:

```
class Point(x : int, y : int)
    var x : int := x
    var y : int := y
    method x() returns int
        return x
    method y() returns int
        return y
    method equal(p : Point) returns bool
        return x = p.x() and y = p.y()
end Point

class ColorPoint(c : string) subclass of Point
    var c : string := c
    method c() returns string
        return c
    method equal(p : Point) returns bool
        return x = p.x() and y = p.y()
    method equal(p : ColorPoint) returns bool
        return x = p.x() and y = p.y() and c = p.c()
end ColorPoint
```

Following the two-level dispatch process described in Section 6.3, when an instance of *ColorPoint* receives the message *equal*, it selects the best matching branch based on the fully evaluated run-time type of the argument expression [BC96, BCC+96]. The first branch specifies the code for arguments of type *Point* and the second branch for arguments of type *ColorPoint* or subtypes thereof [BCC+96]. The second branch is only executed if both the receiver and the argument have a color. In all other cases the color is ignored [MHH91]. It would also be possible to specify additional branches for mixed comparisons as in Example 6.7 [Ghe91].

To validate the above class definitions, the question must be raised whether the type *ColorPoint* really is a subtype of *Point*. Considering the *equal* method in *Point* to be a one-branched multi-method, the associated object types are:

$$\begin{aligned}
Point &\equiv \langle\langle x : unit \to int ; y : unit \to int ; equal : \{Point \to bool\}\rangle\rangle \\
ColorPoint &\equiv \langle\langle x : unit \to int ; y : unit \to int ; c : unit \to string ; \\
&\qquad equal : \{Point \to bool , ColorPoint \to bool\}\rangle\rangle
\end{aligned}$$

According to the subtyping rules for overloaded function types (Section 6.2.1), a multi-method of the type $\{Point \to bool, ColorPoint \to bool\}$ can substitute a multi-method of the type $\{Point \to bool\}$ in any context because the only branch in the latter is type-safely replaced by the first branch in the former. More formally, $\{Point \to bool, ColorPoint \to bool\} \leqq \{Point \to bool\}$ and $ColorPoint \leqq Point$ [BC96, BCC+96].

Intuitively, one might assume that an additional branch is required for every single ancestor in the preceding superclass or supertype chain. For example, if the class *3DColorPoint* were derived from *ColorPoint* then *3DColorPoint*'s *equal* definition would have to be a multi-method with three branches, namely one branch for each of the two preceding superclasses to obtain type-safety and one for *3DColorPoint* itself [BCC+96]. Deep subtyping hierarchies would result in an irreconcilable proliferation of multi-method branches.

Fortunately, the type-safe overriding (in inheritance relationships) or specification (in subtype relationships) of binary methods in terms of encapsulated multi-methods does not require a large amount of extra programming [BCC+96]. It is not necessary to add a branch for every single ancestor [BC96]. In fact, in the case of single inheritance, the number of branches that are sufficient to type-safely define a binary method is independent of both the size and the depth of the hierarchy—it is always equal to two[26] [Cas95, BC96, BCC+96]. Apart from the binary method defined on the type of `self`, it is necessary to add another branch to handle the arguments that could be passed to the method of the supertype:

1. One branch (which is not required for type-safety [Cas95]) describes the "binary part", namely the code for covariantly specialized arguments of the same type as the receiver or a subtype thereof.

2. The second branch is required to establish the substitutability relationship towards the corresponding method of the ancestor[27]. The argument type of this branch must be general enough (meaning sufficiently high in the subtype hierarchy) to handle all remaining arguments not handled by the first branch [BC96]—those arguments whose type lies between the type of the receiver and the original type in which the binary method was first defined [Cas95]. Therefore, the argument type of the second branch is that original type.

Type-safety stems from the fact that the subtyping condition is satisfied: the second branch ensures complete substitutability towards the supertype, while the first branch handles the particular case in which the argument has the same type as the receiver [CL94b, BCC+96].

[26] As noted by [Cas95, BC96, BCC+96], the solution in multiple inheritance is more complicated and may require more branches and additional type-checking constraints.

[27] This extra branch to ensure type-safety of subtyping can be generated automatically, meaning that the technique can be embedded in the technology of existing compilers. These automatically generated branches can "patch" or correct previously ill-typed computations in languages employing type-unsafe covariant specialization such as Eiffel [Mey97] and O_2 [BC96, BCC+96]. Alternatively, similar extensions may allow covariant and contravariant specialization in previously invariant languages like C++ and Java [BC97].

Example 6.10: Binary multi-methods with two branches.
Consider the following chain of subclasses:

```
class 1DVector(x1 : int)
    var x1 : int := x1
    method x1() returns int
        return x1
    method equal(v : 1DVector) returns bool
        return x1 = v.x1()
end 1DVector

class 2DVector(x2 : int) subclass of 1DVector
    var x2 : int := x2
    method x2() returns int
        return x2
    method equal(v : 1DVector) returns bool
        return v.equal(self)
    method equal(v : 2DVector) returns bool
        return x1 = v.x1() and x2 = v.x2()
end 2DVector

class 3DVector(x2 : int) subclass of 2DVector
    var x3 : int := x3
    method x3() returns int
        return x3
    method equal(v : 1DVector) returns bool
        return v.equal(self)
    method equal(v : 3DVector) returns bool
        return x1 = v.x1() and x2 = v.x2() and x3 = v.x3()
end 3DVector
```

For these class specifications to be valid, the associated object types must be in the subtype relationship $3DVector \leqq 2DVector \leqq 1DVector$. In both *2DVector* and *3DVector*, the first branch of the encapsulated multi-method *equal* can replace the corresponding method in the supertype, thus guaranteeing substitutability between these multi-methods. The second branch represents the binary part, namely the covariantly specialized redefinition for arguments of the same type as the receiver.

Notice how the implementation of the first branch is simplified by resending the message with reverted argument and receiver [BC96, BCC+96]. When the first branch is invoked, the receiver "knows" its own type and that the type of the argument is a true supertype of its own type. Therefore, the reverted message is sent to the new receiver with the argument `self` that has a subtype of the new receiver's type, meaning that in any case the second branch of the receiver's *equal* multi-method will be invoked next (except in the trivial case in which the receiver is an instance of *1DVector*).

In general, encapsulated multi-methods with two branches are sufficient to satisfy subtyping even if contravariance and covariance are used together [Cas95, BC96, BCC+96]. To illustrate this formally, consider the following general assumptions:

- Assume we have the types $S_1, \ldots, S_n$ which are supposed to satisfy the subtype relationships $S_n \leqq \cdots \leqq S_1$.
- Assume we want to define for each type $S_1, \ldots, S_n$ a binary method m that returns a result of type $T_1, \ldots, T_n$, respectively[28]. The subtyping rules for function types require that these result types stand in the subtype relationships $T_n \leqq \cdots \leqq T_1$.

These binary methods can type-safely be implemented as encapsulated multi-methods in which the desired branch of type $S_i \rightarrow T_i$ is supplemented by an appropriately chosen branch of type $S_1 \rightarrow T_{i-1}$ to ensure substitutability. According to the subtyping rules for overloaded function types, we have the following type inequalities:

$$\{S_n \rightarrow T_n, S_1 \rightarrow T_{n-1}\} \leqq \cdots \leqq \{S_{i+1} \rightarrow T_{i+1}, S_1 \rightarrow T_i\} \leqq \\ \{S_i \rightarrow T_i, S_1 \rightarrow T_{i-1}\} \leqq \cdots \leqq \{S_1 \rightarrow T_1\}$$

The branch of type $S_1 \rightarrow T_{i-1}$ can replace both branches of the supertype, because for all $i \in \{2, \ldots, n\}$: $S_1 \rightarrow T_i \leqq S_i \rightarrow T_i$ and $S_1 \rightarrow T_i \leqq S_1 \rightarrow T_{i-1}$. This proves that two branches always suffice for binary methods [BC96, BCC+96].

[28]Off course, the binary methods may also be "binary" in the result type. In this special case we have $S_1 = T_1, \cdots, S_n = T_n$.

Chapter 7

Subtyping and Inheritance in C++ and *Susi*

Most of the theoretical aspects discussed so far are, in some way or the other, present in many contemporary programming languages. In this section we will investigate how these theoretical issues are realized or can at least be simulated in two different object-oriented languages, namely C++ as a representative strongly-typed language, and *Susi* [Kur99] as an exploratory dynamically-typed language. We will compare the presented theory with both languages and conclude with a tabular overview of the supported features. For a more detailed view of C++ refer to [Eck95, Str97]. *Susi* is described in Appendix A. While the examples in this chapter still aim at clarifying certain concepts, we will frequently code the same example in both C++ and *Susi* to provide a more comprehensive comparison between the two languages.

7.1 Types and Implementations

One of the most intriguing differences between C++ and *Susi* is that C++ is (mostly) strongly-typed and type checking is performed at compile-time, while *Susi* is dynamically-typed and type checking must be postponed until run-time:

- Section 2.1 explained that in *Susi*, as well as in other Smalltalk derivatives, the compiler neither assigns compile-time types to expressions and objects nor are variables constrained by types—it is syntactically legal to assign any object to any variable. In this sense, it is also syntactically legal to send any message to any object. Erroneous message sends are caught dynamically, leading to the well-known “message not understood” errors at run-time. Nevertheless, *Susi* is not an untyped language, because, due to the lack of “types” in untyped languages, the notion of “type error” would be nonsensical in such a language. Consequently, this notion of type error suggests that *Susi* is indeed a typed language, albeit type checking is performed at run-time. Although not explicitly specified, *Susi* has *implicit types* (Section 2.1):
  - Every object has an implicit object type, namely the set of messages that can type-safely be sent to that object. Since all instances of a class support the same set of messages, it can be said that every class defines an implicit object type.

  - Every variable and formal parameter is constrained by an implicit object type, namely the set of messages sent to the referenced objects via that variable or formal parameter.
  - Although a method signature consists only of a name and an arity, every method has an implicit function type, constructed from the implicit type constraints of that method's formal parameters (Section 5.3.2). *Susi* supports a limited form of functional programming by allowing code blocks to be treated as normal objects which can be passed as message arguments or assigned to variables.

- As explained in Section 2.1, in C++ a compile-time type is assigned to every variable, formal parameter, and expression[1]. Although, due to subtype polymorphism, the run-time type of the expression value may actually be a subtype of the computed compile-time type, the weak principle of substitutability (Section 2.2.1) allows the compiler to statically ensure the absence of run-time type errors.

  - A class explicitly specifies the type of its instances, consisting of the public methods and variables supported by the class instances. In particular, the type does not include private or protected variables and methods. The inclusion of public variables into the type can be expressed in our established model of object types by representing the variable as two methods: an *accessor* function to read the state of the variable and a *mutator* procedure to change the state of the variable [Eck95]. Assume a variable $x : \sigma$ is expressed by the functions $x : \sigma \rightarrow \mathit{unit}$ and $x : \mathit{unit} \rightarrow \sigma$. If the type constraint of x is changed in a subtype to τ, then x is modeled as $x : \tau \rightarrow \mathit{unit}$ and $x : \mathit{unit} \rightarrow \tau$. The contravariant and covariant rules for argument and result types require $\sigma \leqq \tau$ and $\tau \leqq \sigma$, respectively, which leads to $\sigma = \tau$. In other words, the type constraint of a variable may not change in the subtype. In C++ the redefinition of a variable with a modified type constraint in a subclass is regarded as a completely new variable declaration.
  - Variables and formal parameters are explicitly constrained by classes. Class-based conformance (Section 5.1.2) guarantees the absence of run-time type errors.
  - Functions have explicit function types. C++ supports higher-order functions (Section 2.2.3), meaning that (pointers to functions) can be passed as arguments to other functions in the style of functional programming languages.

In *Susi* and C++, objects are defined in terms of class specifications. However, classes in *Susi* bear closer resemblance to our notion of implementations introduced in Section 3.1 and are fundamentally different to classes as in C++, which correspond to our notion of classes presented in Section 5.1.1:

- A class `C` in *Susi* implements an implicit object type σ if every instance of `C` belongs to σ. According to subtype polymorphism (Section 2.2.2), this is the case when the implicit object type τ of the instances of `C` is a subtype of σ. Since the subtyping relationship between implicit object types is structural, so is the relationship between classes and

[1]As already noted, whether these types are inferred by the compiler or specified by the programmer is insignificant.

the implicit object types they implement (Section 3.2). Although a class identifies the implicit object type of its instances, the reverse is not true: the implicit type of an object does not identify the object's class.

- In C++, a type system is automatically associated with the set of classes (Section 5.1.1). There is a one-to-one correspondence between types and implementations. This implements relationship is explicitly stated in terms of class declarations. A class defines an object type and simultaneously the implementation of its elements, meaning that every type has exactly one implementation (sections 5.1.1 and 5.3). Given an object x, the type of x identifies the internal implementation of x (and vice versa).

Example 7.1: `Square` in C++.
Consider the following class specification in C++[2], which specifies an object type and the implementation of its elements:

```
class Square {
public:
  Square(int x) : side(x) {}
  int area() {return side * side;}
  bool equal(Square other) {return side == other.side;}
  int side;
};
```

The first method is the *constructor* used at the instantiation to initialize the object state. Therefore, this method does not belong to the instances of `Square` but rather to the class itself.

- The instance variable `side` is constrained by `int`.
- The formal parameter `other` is constrained by `Square`, meaning that at run-time the actual parameter may belong to a subclass of `Square`.
- The function type of the method `equal` is $Square \rightarrow bool$.

As previously mentioned, the public instance variable `side` can be modeled in terms of an accessor and a mutator function. In this sense, all instances of the class `Square` have the following object type:

$$\langle\langle side : int \rightarrow unit; side : unit \rightarrow int; area : unit \rightarrow int; equal : Square \rightarrow bool\rangle\rangle$$

Example 7.2: `Square` in *Susi*.
Compare the previous class specification in Example 7.1 to the following class definition in *Susi*:

[2] To shorten and simplify the presentation we choose to ignore the issue of *operator overloading* in C++. The function `equal` would typically be replaced by a redefinition of the binary comparison operator `==` [Eck95, Str97].

```
class Square extends Object {
  method new {self.error("does not implement new:0")}
  method new(anInt) {return super.new.side(anInt)}
instance [ side ]
  method side() {return side}
  method side(anInt) {side => anInt}
  method area() {return side.multiply(side)}
  method equal(aSquare) {return side.equal(aSquare.side)}
}
```

The class `Square` inherits from `Object` and adds the specified definitions. The definitions prior to the keyword `instance` specify class variables and methods, while those after `instance` specify the instance variables and methods. The class method `new:0` overrides the inherited method to produce a run-time error, indicating that `Square` can only be instantiated with initialization parameters. The creation method `new:1` generates a new instance by bypassing the redefinition of `new:0` and calling the inherited version via `super`. This inherited version creates an empty instance which is then initialized by setting the side length with the instance method `side:1`. Instances of `Square` also have methods to query a square's side and area, as well as a method for comparing squares to each other.

- The instance variable `side` is implicitly constrained by the object type corresponding to the set of all messages sent to `side` in the methods of `Square`[3]:
  $\langle\langle \mathit{multiply} : \langle\langle\rangle\rangle \to \langle\langle\rangle\rangle ; \mathit{equal} : \langle\langle\rangle\rangle \to \langle\langle\rangle\rangle \,\rangle\rangle$
- The formal parameter `anInt` is implicitly constrained by $\langle\langle\rangle\rangle$. The formal parameter `aSquare` is implicitly constrained by $\langle\langle \mathit{side} : \mathit{unit} \to \langle\langle\rangle\rangle \,\rangle\rangle$.
- The implicit function type of the method `equal` is $\langle\langle \mathit{side} : \mathit{unit} \to \langle\langle\rangle\rangle \,\rangle\rangle \to \langle\langle\rangle\rangle$.

Although the empty object type $\langle\langle\rangle\rangle$ occurs rather frequently, these implicit types give a valuable indication of the required messages. In summary, all instances of `Square` have the following implicit object type:

$$\langle\langle \mathit{side} : \mathit{unit} \to \langle\langle\rangle\rangle ; \mathit{side} : \langle\langle\rangle\rangle \to \langle\langle\rangle\rangle ; \mathit{area} : \mathit{unit} \to \langle\langle\rangle\rangle ;\\ \quad \mathit{equal} : \langle\langle \mathit{side} : \mathit{unit} \to \langle\langle\rangle\rangle \,\rangle\rangle \to \langle\langle\rangle\rangle \,\rangle\rangle$$

Example 7.3: `PolarPoint` and `CartPoint` in *Susi*.
In *Susi*, several classes can implement the same implicit object type. For example, the implementations from Example 3.7 can be coded in *Susi* as:

```
class PolarPoint extends Object {
  method new {self.error("does not implement new:0")}
  method new(x, y) {return super.new.init(x, y)}
instance [ rho theta ]
  method init(x, y) {
      rho => x.multiply(x).add(y.multiply(y)).sqrt;
      theta => y.divide(x).atan}
  method x {return rho.multiply(theta.cos)}
```

[3]Since no constraints are visible on the argument and result types of `multiply` and `equal`, we assume them to be constrained by the empty object type $\langle\langle\rangle\rangle$.

```
  method y {return rho.multiply(theta.sin)}
  method equal(aPoint) {
      return self.x.equal(aPoint.x).and(self.y.equal(aPoint.y))}
}

class CartPoint extends Object {
  method new {self.error("does not implement new:0")}
  method new(x, y) {return super.new.init(x, y)}
instance [ x y ]
  method init(aFloat, anotherFloat) {x => aFloat; y => anotherFloat}
  method x {return x}
  method y {return y}
  method equal(aPoint) {
      return self.x.equal(aPoint.x).and(self.y.equal(aPoint.y))}
}
```

Both these classes implement the following implicit object type (ignoring the method `init`) which is similar to *Point* from Example 2.18:

$$\langle\langle x : unit \rightarrow \langle\langle\rangle\rangle; y : unit \rightarrow \langle\langle\rangle\rangle; equal : \langle\langle x : unit \rightarrow \langle\langle\rangle\rangle; y : unit \rightarrow \langle\langle\rangle\rangle\,\rangle\rangle \rightarrow \langle\langle\rangle\rangle\,\rangle\rangle$$

7.1.1 Subtyping

While in C++ the subtype hierarchy is embedded in the class hierarchy, in *Susi* the implicit function and object types can be arranged in an implicit subtyping (conformance) hierarchy which is independent of the *Susi* class hierarchy:

- The conventional subtyping rules from Section 2.2 can be employed to establish subtype relationships between implicit object and function types, respectively. In a sense, subtyping between implicit types in *Susi* is purely structural and can be verified by the rules described in [Car88, AC93]. However, these subtyping relationships are neither inferred nor checked by the compiler. The subtyping of implicit object types obeys the full motion of contravariant argument types and covariant result types. Due to implicit and structural subtyping, an implicit object type inherently has multiple supertypes (Section 2.2.5).

- In C++, a subtyping hierarchy is superimposed onto the class inheritance hierarchy (Section 5.1.2). Therefore, all direct subtype relationships are named explicitly, accompanied by the inheritance relationships between the associated classes. While C++ offers function types and higher-order functions, it does not support function subtyping. Consequently, the functions that a subclass and its superclass have in common must have identical function types[4]. Similarly, when a function f is passed as an argument to another function, then f's type must be identical to the type constraining the corresponding formal parameter. Argument and result types in C++ are *invariant* and subtyping of object types is shallow (as opposed to deep subtyping in Section 2.2.4).

Example 7.4: `ColorSquare` in *Susi*.
The following class inherits from `Square` (Example 7.2) and overrides the method `equal`:

[4] Clearly, this is only an issue when inherited methods are redefined by the subclass (Section 6.1).

```
class ColorSquare extends Square {
  method new {self.error("does not implement new:0")}
  method new(anInt, aString) {return super.new(anInt).color(aString)}
instance [ color ]
  method color() {return color}
  method color(aString) {color => aString}
  method equal(aColorSquare) {
      return side.equal(aColorSquare.side).and(color.equal(aColorSquare.color))}
}
```

The implicit object type defined by `ColorSquare` is:

$$\begin{aligned}\langle\langle & side : unit \rightarrow \langle\langle\rangle\rangle ; side : \langle\langle\rangle\rangle \rightarrow \langle\langle\rangle\rangle ; area : unit \rightarrow \langle\langle\rangle\rangle ;\\ & color : unit \rightarrow \langle\langle\rangle\rangle ; color : \langle\langle\rangle\rangle \rightarrow \langle\langle\rangle\rangle ;\\ & equal : \langle\langle side : unit \rightarrow \langle\langle\rangle\rangle ; color : unit \rightarrow \langle\langle\rangle\rangle\,\rangle\rangle \rightarrow \langle\langle\rangle\rangle\,\rangle\rangle\end{aligned}$$

The implicit type of `ColorSquare` is not a subtype of the implicit type of `Square` due to the change of the implicit function type of `equal`: the implicit function type of `equal` in `ColorSquare` is not a function subtype of the implicit function type of `equal` in `Square`, because the implicit argument type of the former is a subtype of the implicit argument type of the latter—apparently breaking the contravariant requirements of function subtyping. If `ColorSquare` did not override `equal`, then the implicit type of `ColorSquare` would indeed be a subtype of the implicit type of `Square`.

Example 7.5: `AdvancedSquare` in *Susi*.
The implicit object type of the following class is a subtype of the implicit type of `Square` in Example 7.2:

```
class AdvancedSquare extends Square {
  method new {self.error("does not implement new:0")}
  method new(anInteger) {return super.new(anInteger)}
instance
  method perimeter {return side.multiply(4)}
}
```

Example 7.6: `AdvancedSquare` in C++.
A definition similar to Example 7.4 in illegal in C++, because the subclass *must* also produce a subtype. Example 7.5 can be coded in C++ as:

```
class AdvancedSquare : public Square {
public:
  AdvancedSquare(int x) : Square(x) {}
  int perimeter() {return side * 4;}
};
```

As explained in sections 2.2.2 and 5.1.2, the issues of subtyping and class-based conformance have impacts on the legality of assignment statements (the following points apply equally to parameter passing):

- In *Susi*, assignment statements are not statically verified by the compiler, meaning that any object can be assigned to any variable. However, following [BHJL86], the implicit conformance relationships can informally justify assignment statements in dynamically-typed languages (Section 5.3.2). The implicit type of an object should structurally conform to the implicit type constraint of the variable or formal parameter it is being assigned to. Indeed, this structural conformance is verified at run-time. Assume an object x of implicit object type σ is assigned to a variable a implicitly constrained by a type τ. In accordance with the assignment rules described in Section 2.2.2, run-time type errors may occur if type σ is not an implicit subtype of type τ. If σ does not conform to τ, then x may receive an unanswerable message [BHJL86, Jon92]. Run-time type errors will not occur if $\sigma \leqq \tau$.

- In C++, the variables and formal parameters are constrained by classes. Therefore, the subclass relations determine the legality of assignment statements. The class of the object must be a subclass of the class constraining the variable. This assignment is type-safe because class-based conformance guarantees that the subclass objects are substitutable for superclass objects. A significant difference to the approach in *Susi* is that, while in *Susi* the implicit type constraint of a variable is concerned only with the behavior of the referenced object, in C++ the class constraint restricts the referenced object to a specific implementation.

Example 7.7: Assignment conformance in *Susi*.
While checking the legality of assignment statements in C++ is fairly straightforward, in *Susi* the responsibility is laid on the programmer. Ignoring the implicit subtypes and type constraints may be penalized with run-time type errors. Taking the class definitions from examples 7.2 and 7.4, consider the following fragment of code:

```
[ aSquare aColorSquare ]
aSquare => Square.new(1);
aColorSquare => ColorSquare.new(2, "red");
aSquare.side; aSquare.area; aSquare.equal(aSquare);
aColorSquare.color; aColorSquare.break;
```

The variables `aSquare` and `aColorSquare` are respectively constrained by the implicit object types:

$$\langle\langle side : unit \rightarrow \langle\langle\rangle\rangle ; area : unit \rightarrow \langle\langle\rangle\rangle ; equal : \langle\langle\rangle\rangle \rightarrow \langle\langle\rangle\rangle \rangle\rangle$$
$$\langle\langle color : unit \rightarrow \langle\langle\rangle\rangle ; break : unit \rightarrow \langle\langle\rangle\rangle \rangle\rangle$$

The first assignment is justifiable because the implicit type of the `Square` instance conforms to the implicit type constraint of `aSquare`. However, the implicit type of the `ColorSquare` instance does not conform to the implicit type constraint of `aColorSquare`. Consequently, a run-time type error occurs.

7.1.2 Inheritance

Both C++ and *Susi* view class specifications as the units of code reuse. A class may inherit from another class, meaning that the former shares all the variable and method declarations of the latter. In accordance with the segregation into public, private, and protected attributes, in C++ only public and protected attributes can typically be inherited[5]. C++ and *Susi* support recursive objects via the late-bound pseudo-variables `this` and `self`, respectively, allowing inherited methods to be internalized as indigenous methods by inheritors (Section 4.2). The main difference is that inheritance in *Susi* is used only for implementation sharing, while, due to the superimposed subtype hierarchy, class inheritance in C++ is inevitably connected to subtyping relationships.

- *Susi* supports only single implementation inheritance; the inheritance hierarchy forms a tree (Section 4.1). Example 7.4 should have made clear that inheritance in *Susi* is purely an implementational issue and independent of subtyping[6]. Therefore, inheritance is not restricted to maintain subtyping and methods may be arbitrarily overridden or canceled in subclasses. We shall return to this point in Section 7.2.

- C++ permits multiple class inheritance with appropriate ambiguity resolution strategies [Sny86, Eck95, Str97]; the class hierarchy forms a directed acyclic graph (Section 4.1). As in most strongly-typed object-oriented programming languages, type inheritance (Section 4.4) is coupled to subtyping. Inheritance and method overriding are restricted to maintain subtyping relationships—an issue dealt with in the following section.

7.2 Subtyping vs. Inheritance

Sections 2.1, 2.2, 3.1, and 4.3 presented types, subtyping, implementations, and implementation inheritance as separate and independent notions. While this distinction is preserved in *Susi*, the orthogonality between subtyping and inheritance is not given in C++:

- Following the unification of types and implementations into classes in C++ (by automatically associating a type system with the specified set of classes), the notions of subtyping and class inheritance are collapsed by superimposing a subtype hierarchy onto the class hierarchy (Section 5.1.2). Consequently, the theoretically independent subtype and inheritance relationships are captured by a single relationship between classes. C++ insists that subclasses generate substitutable subtypes, meaning that the type σ of a subclass C and the type τ of the corresponding superclass C' must be related as $\sigma \leqq \tau$. The C++ compiler can statically verify whether the class hierarchy conforms to the semantics of the superimposed type system. The requirement that subclasses generate subtypes and subtypes are generated by subclasses introduces the following hindrances into object-oriented programming in C++:
  - Although types should describe only externally observable behavior, the objects of a subtype cannot be implemented completely different from the objects of the supertype because the class associated with the subtype must inherit from the class associated with the supertype. In other words, a type can only be a subtype

[5] Actually, C++ supports different forms of inheritance, namely public, private, and protected inheritance, specifying the visibility and accessibility of the inherited attributes (in dependence of their visibility in the superclass) in the subclass instances [Eck95, Str97].

[6] Naturally, the notion of implicit object types precludes any form of type inheritance (Section 4.4).

of another type if the associated classes are related by inheritance—their objects must share a common implementation. Applications similar to that illustrated in Example 5.7 cannot be straightforwardly coded in C++.

– In class-based languages, inheritance is typically constrained to maintain the subtype relationship between the associated types. Therefore, overriding is restricted to preserve subtyping (Section 5.2) and the cancellation of methods is forbidden. A class similar to Example 7.4 cannot be implemented in such languages, illustrating how inheritance may be precluded even when substitutability is not an issue. The lack of function subtyping in C++ a priori guarantees that method overriding retains subtyping. Further, C++ offers the possibility of *private* inheritance, an implementation detail that is not reflected in the type of the subclass [Str97]—the subtype relation to the superclass is lost (Section 5.3). The limitation of private inheritance is that all inherited attributes become part of the hidden (private) implementation in the subclass and are therefore removed from the object type. In this sense, private inheritance is equivalent to composition [Str97] and Example 7.4 still cannot be meaningfully coded in C++.

In effect, typical C++ programmers elide the notions of types and implementations and only think in terms of classes and subclassing, exhibiting a clear deficiency in understanding vital programming concepts [LP91]. However, the frequent correspondence between types and implementations may actually justify this approach in practice: the combined hierarchy represents a successful and widely-accepted economy of concepts which entails a simplification for a wide variety of applications and hierarchies.

- Implementations and implicit types in *Susi* are separate and autonomous concepts related solely by the structurally inferred implements relationship. Inheritance in *Susi* is only an implementation technique, facilitating a clear differentiation between the explicitly specified inheritance hierarchy used for the sharing of class specifications and the structurally inferred, implicit subtype relationships. This allows the *Susi* programmer to design in terms of types and subtypes and to implement in terms of classes and inheritance hierarchies. As investigated in Section 5.3.2, a subclass does not necessarily produce subtype objects and a subtype is not necessarily implemented by a subclass:

  – Assuming a class C implements the implicit object type σ and a subclass C' implements the implicit object type τ, then, although C and C' are related by inheritance, in *Susi* instances of C' are not generally substitutable for instances of C because τ is not necessarily a subtype of σ. As explained in Section 5.3.2 and illustrated in Example 7.2, a subclass in *Susi* may redefine an otherwise inherited method to issue a run-time error, effectively removing that method from the implicit object type defined by the class. Further, sections 5.3.2 and 6.1.1 and Example 7.4 showed that a subclass may arbitrarily redefine the implicit function type of an inherited method, potentially disobeying the rules for function subtyping (Section 2.2.3). In both these cases the subtype relationship between the implicit object types associated with the classes may be lost. Defiantly programming in terms of these incorrect subtype relationships will probably lead to run-time errors. Inheritance without subtyping means that unrelated types may have related implementations (examples 7.2 and 7.4).

– Assuming a class C implements the implicit object type σ and a class C' implements the implicit subtype τ, then, although σ and τ are related by subtyping, in *Susi* objects of σ and τ need not share a common implementation because C' is not necessarily a subclass of C. This follows the spirit of types and subtyping (sections 2.1 and 2.2) in the sense that substitutability is concerned only with an object's behavior and not its implementation. An example follows:

Example 7.8: Subtyping without inheritance in *Susi*.
Looking back at Example 2.4, in *Susi* we can implement the types *Stack* and *Bag* as follows:

```
class Bag extends Object {
  method new {return super.new.init}
instance [ storage index ]
  method init {storage => Array.new(100); index => 1}
  method put(anObject) {
    index.greater(storage.size).ifTrue({storage=>storage.expand(100)});
    storage.indexPut(index, anObject);
    index=>index.add(1)}
  method get {[ temp ]
    index=>index.subtract(1);
    temp => storage.index(index);
    return temp}
}

class Link extends Object {
  method new {self.error("does not implement new:0")}
  method new(object, next) {return super.new.init(object, next)}
instance [ object next ]
  method init(anObject, aLink) {object => anObject; next => aLink}
  method object {return object}
  method next {return next}
}

class Stack extends Object {
  method new {return super.new}
instance [ root ]
  method put(anObject) {root => Link.new(anObject, root)}
  method get {[ temp ] temp => root.object; root => root.next; return temp}
  method dup {[ temp ] temp => self.get; self.put(temp); self.put(temp)}
}
```

The implementation `Stack` uses the previously defined class `Link`. While the implicit object type implemented by `Stack` is a subtype of the implicit object type implemented by `Bag` (ignoring the method `init`), the classes `Stack` and `Bag` are not related by inheritance. Example 5.9, demonstrating subtyping and inheritance hierarchies running in opposite directions, could be similarly implemented in *Susi*.

7.2.1 Simulation via Abstract Classes in C++

Section 5.3.1 investigated the notion of abstract classes, allowing the restrictions pertaining to class-based languages such as C++ to be circumvented to a certain degree. In particular, abstract classes provide the possibility to specify multiple implementations of a single object type and to express subtyping without inheritance.

A class in C++ is abstract if the class declaration has at least one *pure virtual function.* A pure virtual function is a function whose body has been replaced by `=0`. This function must be redefined by a subclass. Only objects of the concrete subclass can be instantiated.

Example 7.9: Separating subtyping and inheritance in C++.
To illustrate the separation of subtyping and inheritance in C++ via abstract classes, we will recall Example 7.3 and implement the abstract class `Point`, representing a subtype of `SimplePoint`, and the concrete implementations `PolarPoint` and `CartPoint`:

```
class SimplePoint {
public:
  virtual double x()=0;
  virtual double y()=0;
};

class Point : public SimplePoint {
public:
  virtual bool equal(Point*)=0;
};

class PolarPoint : public Point {
public:
  PolarPoint(double x, double y) : rho(sqrt(x*x+y*y)), theta(atan(y/x)) {}
  virtual double x() {return rho*cos(theta);}
  virtual double y() {return rho*sin(theta);}
  virtual bool equal(Point* p) {return (x()==p->x())&&(y()==p->y());}
protected:
  double rho, theta;
};

class CartPoint : public Point {
public:
  CartPoint(double x, double y) : i(x), j(y) {}
  virtual double x() {return i;}
  virtual double y() {return j;}
  virtual bool equal(Point* p) {return (x()==p->x())&&(y()==p->y());}
protected:
  double i, j;
};
```

`SimplePoint` defines an abstract interface and `Point` a subtype thereof. The interface `Point` is implemented by the concrete subclasses `PolarPoint` and `CartPoint`, employing completely unrelated internal implementations. Concrete implementations of the interface `SimplePoint` would be coded as direct subclasses of `SimplePoint`.

- The subclass relationship between fully abstract classes corresponds to a subtype relationship between types. Subtyping is not restricted to implementation inheritance because these abstract classes have no implementations. The relationship between `SimplePoint` and `Point` corresponds to a named subtype (and type inheritance) relationship.

- The subclass relationship between fully or partially implemented classes represents the sharing of code. However, this inheritance mechanism is still bound to subtyping.
- The subclass relationship between an abstract class and a concrete class corresponds to a named implements relationship between an object type and an implementation. The abstract class `Point` defines an interface with the multiple implementations `PolarPoint` and `CartPoint`, respectively.

Instead of identifying the subtyping and inheritance hierarchies into a single hierarchy, programming with abstract classes "mixes" the two hierarchies into one, meaning that parts of the hierarchy (the abstract classes) represent types and subtyping while other parts (the concrete subclasses) represent implementations. The subclass relationship represents subtyping, inheritance, or the implements relationship, depending on whether the related classes are abstract or concrete.

Example 7.10: Abstract classes and subtype polymorphism.
As mentioned in Section 5.3.1, abstract classes allow the programmer to abstract away from implementations and return to programming in terms of types, by defining variables as pointers to instances of abstract classes. For example, if a variable `x` is declared as a pointer to an instance of `Point`, then subtype polymorphism allows the variable to refer to instances of the concrete subclasses `PolarPoint` and `CartPoint`. Thus, the declaration of `x` now correctly constrains the behavior rather than the implementation of the objects referenced by `x`:

```
Point* x = new PolarPoint(3, 4);
Point* y = new CartPoint(5, 6);
if (x->equal(y))   // concrete implementations of objects referenced by
  printf("true");  // x and y are unknown, only interfaces are known
```

7.3 Contravariance vs. Covariance

In Section 6.1.1, we noted that the issues of covariant and contravariant specialization are not confined to strongly-typed languages, but are equally present in dynamically-typed languages like *Susi*. In this section we will investigate the possibilities for covariant specialization and multiple dispatching in C++ and *Susi*, concluding that the situation in *Susi* is not quite so fundamentally different to C++, even though *Susi* allows methods to be covariantly overridden in subclasses (sections 5.2, 5.3.2, and 7.2):

- Subtyping of implicit object types in *Susi* obeys the conventional subtyping rules for function and object types (sections 2.2.3 and 2.2.4). A subtype may have contravariantly specialized (implicit) argument types and covariantly specialized (implicit) result types. *Susi* programmers must respect the same constraints as apply to strongly-typed languages like Trellis/Owl [SCB+86], even though (or, rather, exactly because) type checking is performed at run-time (Section 5.3.2). As shown in Example 7.4, if a subclass covariantly overrides an inherited method, then the subtyping relationship between the implicit object types defined by the classes must be abandoned; otherwise run-time type errors will occur. In terms of the tradeoff described in Section 6.1.1, *Susi* supports subtype polymorphism and does not forbid covariant specialization at the price of compile-time type checking. Type-safe covariant specialization cannot be realized as in Section 6.3.1 because *Susi* is a single-dispatching language without multi-methods[7]. The notion of overloading in *Susi* (Section A.3.3) is profoundly different to the notion of overloaded functions in Section 6.2.1. Overloading in *Susi* can be characterized as *overloading on arity*, meaning that the same method name can be used for *distinguished* methods within a class specification (Example 7.2). In this sense, overloading in *Susi* is a syntactic issue with absolutely no impact on run-time method lookup.

- Subtyping in C++ in invariant[8], impeding both the covariant and type-safe contravariant specialization (sections 2.2.4 and 7.1.1) allowed in Trellis/Owl [SCB+86]. In terms of the tradeoff described in Section 6.1.1, the type system of C++ precludes covariant specialization by the presence of compile-time type checking and subtype polymorphism. As with *Susi*, C++ does not support late-bound overloaded functions as described in Section 6.2.1; method lookup is based only on the class of the message receiver (Section 6.2.2). While the application of overloaded functions in Section 6.2.1 is resolved at run-time, overloading in C++ is fully resolved at compile-time (Section 6.2.2), based on the first step of the selection discipline presented in Section 6.2.3.

In the following section, we will show how the type-safe binary methods from Section 6.3.1 can be realized in C++ and *Susi*, by simulating the multiple dispatching nature of encapsulated multi-methods (Section 6.3).

7.3.1 Simulating Multiple Dispatching in C++ and *Susi*

Lacking the notion of encapsulated multi-methods in C++ and *Susi*, the mechanism can be simulated in terms of the conventional dispatch model based on a single message receiver[9] [Cha92]. The approach is general enough to be applicable to C++, *Susi*, and other object-oriented languages as well [Ing86], facilitating type-safe covariant specialization in single-dispatching programming languages like Java [GJS96] or Eiffel [Mey97].

In languages like C++ and *Susi*, the process of sending a message to a receiver reduces the polymorphism of that receiver: while the class of the receiver is not known before the message send, the method lookup inherently reveals the receiver's class. In other words, when an object's method is being executed as the response to a message send, the type of `self` is known inside the method—the object "knows" itself and its type.

This resolution of polymorphism need not be confined to the receiver alone. Multiple dispatching can be accomplished in a single-dispatching language as a sequence of multiple

[7]Interestingly, the inclusion of encapsulated multi-methods was considered in the initial design of *Susi*, but the idea was later discarded due to implementational complexities.

[8]Redefining an inherited function with a different function type in a subclass does not override but *overloads* the inherited function. Both the inherited and the new definition are present in the subclass. The appropriate definition is selected at compile-time (Section 5.2).

[9]In fact, the compiler extensions mentioned at the end of Section 6.3.1 can be implemented without changing the run-time semantics of a language. Instead, programs using multi-methods can be automatically translated at source level into programs that do not [BC96, BC97], simulating multi-methods as described in this section [BCC+96].

message sends, relaying the message to each argument in turn. Each of these single-dispatched message sends reveals the class of an argument object of the original message, effectively resolving the polymorphism of that argument [BCC+96]. The progress of this simulated dispatch process can be coded into the current message name [Ing86] through appropriate name-mangling [BC97]:

- Assume the message m with arguments $x_1, \ldots, x_n$ of classes $C_1, \ldots, C_n$ is sent to an object x of class C. At run-time the method m defined in C will be executed.

- Since m in x "knows" the class C of x, this class is coded into the previous message name, for example producing mC. This message is now sent to the first argument x_1 with the original receiver x and the remaining $x_2, \ldots, x_n$ as arguments. The class C_1 of x_1 must have a method mC, which is executed next.

- In turn, each argument x_i receives a message which it processes by coding its own class C_i into the current message name and relaying that new message to the next argument.

- The last argument x_n will receive a message whose name reveals the classes of all preceding arguments. Therefore, x_n must have methods whose names reflect all possible encodings corresponding to $C, C_1, \ldots, C_{n-1}$.

In other words, this technique makes each argument, in turn, the receiver of a message which encodes the classes of the already "visited" arguments [Budd91]. Each message send reduces the polymorphism of the previous message by one [Ing86].

Since we are interested in implementing binary methods, we will reduce our discussion to double dispatching where a message with a single argument is sent to a receiver. The first message to the receiver only does half the job since the class of the argument remains unknown. A second message to the argument resolves this remaining polymorphism [Ing86]. The information about the first receiver's class is coded into the second message name[10].

Example 7.11: `Point` and `ColorPoint` in *Susi*.
The encapsulated multi-methods in Example 6.9 can be simulated in *Susi* as follows [BC96]:

```
class Point extends Object {
  method new {self.error("does not implement new:0")}
  method new(x, y) {return super.new.init(x, y)}
instance [ x y ]
  method init(aFloat, anotherFloat) {x => aFloat; y => anotherFloat}
  method x {return x}
  method y {return y}
  method equal(p) {
    return p.equalPoint(self)}
  method equalPoint(p) {
    return x.equal(p.x).and(y.equal(p.y))}
  method equalColorPoint(cp) {
    return x.equal(cp.x).and(y.equal(cp.y))}
}
```

[10] This approach is also known as *double dispatching* in Smalltalk-80 [GR83, BCC+96] or, more generally, as the *visitor pattern* [GHJV95], because it consists of subsequently visiting each argument.

```
class ColorPoint extends Point {
  method new {self.error("does not implement new:0")}
  method new(x, y, c) {return super.new(x, y).init(c)}
instance [ c ]
  method init(aString) {c => aString}
  method c {return c}
  method equal(p) {
    return p.equalColorPoint(self)}
  // equalPoint is inherited from Point
  method equalColorPoint(cp) {
    return x.equal(cp.x).and(y.equal(cp.y)).and(c.equal(cp.c))}
}
```

When the message `p1.equal(p2)` is sent, then the definition of `equal` in `Point` or `ColorPoint` is executed, depending on the class of `p1`. This first message resolves the class of `p1`. The executed method `equal` encodes `p1`'s class into a new message name `equalPoint` or `equalColorPoint` and resends this message to `p2`, passing `p1` as an argument. This second message resolves the class of `p2`. The method `equalPoint` or `equalColorPoint` in `p2`'s class is executed, which, knowing the class of `p2` and using the information about `p1`'s class encoded in the message name, can consider or ignore the color in the comparison. The color is only considered if both `p1` and `p2` are instances of `ColorPoint`.

Example 7.12: `Point` and `ColorPoint` in C++[11].

```
class Point {
public:
  Point(double i, double j) : x(i), y(j) {}
  virtual bool equal(Point* p) {
    return p->equalPoint(this);}
  virtual bool equalPoint(Point* p) {
    return (x==p->x)&&(y==p->y);}
  virtual bool equalColorPoint(ColorPoint* cp) {
    return (x==cp->x)&&(y==cp->y);}
  double x, y;
};

class ColorPoint : public Point {
public:
  ColorPoint(double i, double j, int n) : Point(i, j), c(n) {}
  bool equal(Point* p) {
    return p->equalColorPoint(this);}
  // method equalPoint(Point) is inherited from Point
  bool equalColorPoint(ColorPoint* cp) {
    return (x==cp->x)&&(y==cp->y)&&(c==cp->c);}
  int c;
};
```

[11] Typical C++ compilers will not accept the following definitions. It is usually necessary to provide an empty class definition of `ColorPoint` before the definition of `Point` to allow the definition of `ColorPoint` pointers. Similarly, the body of `Point::equalColorPoint` must usually be defined *after* the definition of `ColorPoint`, because it requires `ColorPoint` to already be defined.

7.4 Summary

The following table provides an overview of the similarities and differences of C++ and *Susi*:

Feature	**C++**	***Susi***
Type checking	compile-time	run-time
Variable is constrained by	class	implicit object type (set of methods sent via variable), no static constraints
Assignment conformance based on	class hierarchy (checked at compile-time)	implicit subtype hierarchy (conformance check at run-time)
Reverse assignments	• (`dynamic_cast`)	• (due to lack of static constraints)
Function types	•	• implicit type constraints of arguments
Subtyping function types	○	•
Higher-order functions	•	•/○ limited form via `Block` objects
Object types	set of public functions and variables	set of methods supported by object
Principle of substitutability	weak	weak
Subtyping object types	named verified at compile-time	implicit and structural verified dynamically
Deep/shallow subtyping	shallow	deep
Multiple subtyping	•	• (consequence of structural subtyping)
Recursive types	•	•
Recursive objects	• supported via `this`	• supported via `self`
Recursive implementations	can be simulated via wrapper methods	supported via `self class`
Implementation unit	class (combined with type)	class (pure implementation)
Separation of type and implementation	•/○ simulation via abstract classes	•
Multiple implementations of type	•/○ simulation via abstract classes	•
Implements relationship	named	structural (based on implicit subtyping)
Symmetry of implements relationship	class $\Rightarrow$ type, type $\Rightarrow$ class	class $\Rightarrow$ type, type $\not\Rightarrow$ class
Inheritance hierarchy	directed acyclic graph (multiple)	tree (single)
Partial inheritance	private attributes are not inherited	cancel method by issuing run-time error
Method overriding	only body (invariant function types)	arbitrary change implicit function type
Type inheritance	coupled to subtyping	○
Subtyping without inheritance	•/○ simulation via abstract classes	•
Inheritance without subtyping	•/○ limited via private inheritance	•
Strict encapsulation	○	•
Granularity of encapsulation	class	object
Abstract classes	•	• by convention and virtue of late binding
Contra-/covariant specialization	•/○ simulation via abstract classes	•
Overloading	• resolved at compile-time	•/○ on arity, resolved at compile-time
Encapsulated multi-methods	○ can be simulated as message sequence	○ can be simulated as message sequence

Chapter 8

Conclusions

The intent of this study was to present a clear and uniform view of subtyping and inheritance in object-oriented programming and practice. We conclude that a consistent interpretation of types is necessary to provide a clear distinction between subtyping and inheritance. We assign an abstract but precise meaning to types in terms of predicates. This interpretation model is general enough to encompass languages that separate types from implementations as well as languages that do not. Our informal notion of predicates may describe arbitrary object properties, ranging from requiring a set of supported messages to implementational constraints.

The abstract definition of types is be used as a sound basis for a general definition of subtyping, specified in terms of a relationship between the associated predicates. By formulating concrete predicates describing sets of functions and objects, respectively, we specialize the abstract notion of types to function and object types. Consequently, we derive the concrete subtyping relationship between function and object types, based on the general definition of subtyping in terms of logical implication. This approach is to be preferred over other approaches, which directly define the concrete subtyping relationship for function and object types, and explain it retrospectively.

We further conclude that types and subtyping exist in dynamically-typed object-oriented programming languages. In such environments it is particularly important to have a founded understanding of object types and subtyping, since the entire responsibility for avoiding type errors is laid upon the programmer. An implicit object type can be associated with every implementation, and these implicit object types can be related in an implicit subtyping hierarchy. This subtyping hierarchy is quite distinct from the explicit implementation hierarchy. Similarly, we investigated how a variable can be associated with implicit type constraints consisting of the set of messages sent via that variable. The implicit type constraints of formal parameters lead to the description of implicit function types associated with methods. Informally programming in terms of these implicit object and function types and variables with implicit type constraints leads to robuster and more reliable software. Moreover, these implicit types should be made an integral part of class documentation and should be exhibited in the names of variables.

The precise separation between types and implementations, related solely by the implements relationship, has proved beneficial. Types and implementations represent different views of a program, and as such should be kept apart conceptually. This distinction goes in the spirit of abstract data types and enforces the principle of modularity.

The design of an object-oriented system will typically produce an explicit subtype hierarchy. In this sense, the subtype relationships are effectively part of the design and should not be determined by subtle inference mechanisms. We conclude that named subtyping provides a higher level of abstraction than structural subtyping, making the type name an integral part of the described object behavior. The reconciliation of named and structural subtyping into a hybrid system would bear some of the flexibility of dynamically-typed languages and the safety and legibility of explicitly specified subtyping.

We obtain a more uniform view of implementation and type inheritance by generalizing inheritance as a mechanism for sharing abstract and recursive program attributes. In fact, our abstract notion of inheritance is general enough to describe both the traditional form of inheritance found in class-based languages, as well as the notion of delegation found in prototype-based languages [Lie86, US87]. While subtyping and inheritance are frequently introduced hand in hand, our definition of inheritance independently from subtyping promotes the view of separate concepts. We conclude that inheritance is an essential mechanism for structuring large programs, and necessary to produce maintainable, reusable, and flexibility software. The benefits of replacing inheritance by mechanisms like composition remain to be questioned.

Further, we show that the strict identification of subtyping and inheritance compromises the expressiveness of strongly-typed object-oriented programming languages, prematurely excluding certain opportunities for code reuse and exposing implementation details in the subtyping relationships. These deficiencies can be effectively circumvented in class-based languages supporting the mechanism of abstract classes. The class hierarchy interleaves the subtyping and inheritance relationships. Abstract classes do not solve the problem of inheritance without subtyping.

Finally, we show that although contravariant specialization is commonly regarded as counter-intuitive, covariant specialization as implemented in Eiffel is just as unnatural, equally precluding the interaction of seemingly compatible objects. While the problem for the type system can be solved by separating subtyping from inheritance, the problem for the client can only be solved by a reconsideration of the run-time semantics of message passing. The mechanism of encapsulated multi-methods preserves the subtyping and substitutability between covariantly specialized objects, providing an intuitive and natural solution. The abstract interpretation of types can be mapped to overloaded functions and subtyping, providing the desired level of consistency and uniformity in the overall presentation. We illustrate how encapsulated multi-methods can be simulated in both dynamically-typed and strongly-typed programming languages by sequences of normal message sends.

We conclude that the separation of types, implementations, subtyping, and inheritance is fundamental both in class-based languages like C++ that identify types and implementations, as well as in dynamically-typed languages like *Susi* without any explicit notion of type. Abstract classes in C++ and dynamic typing in *Susi* permit the design in terms of object behaviors to be distinguished from the implementations of these behaviors.

Appendix A

Susi

The programming language *Susi* [Kur99] aims to be a pure object-oriented programming language. *Susi* is the product of the attempt to design and implement a programming language based on the main paradigms of object-orientation and on a high level of uniformity. *Susi* has elements of widely spread object-oriented languages such as C++ [Str97] and especially Smalltalk-80 [GR83]. The main design issue was to keep the language syntax and semantics at an absolute minimum, while at the same time offering a full-fledged language for object-oriented programming.

Many explicit features of common object-oriented languages such as virtual functions, templates, and run-time type identification are either implicitly included or made obsolete in *Susi* due to various characteristics such as type-unbound variables and dynamic method dispatching. Explicit storage deallocation is also made obsolete due to the presence of a garbage collector.

A.1 Basic Concepts

A.1.1 Objects and Messages

Objects provide the basic means of abstraction in *Susi*. Objects represent various software system components. These may be simple numbers and strings or more complex components such as files, directories, pointing devices (mouse) and graphical displays. An object is defined by a set of private *variables* and a set of *operations* which manipulate the variables. Different objects have different operations. Objects representing directories may provide operations to make and remove directories, while objects representing pointing devices may provide operations to retrieve the current pointer position or to change the cursor.

Messages provide the basic concept of interaction between objects. Objects receive messages as requests to perform one of their operations. The message *receiver* decides which operation corresponds to the message and executes it. *Methods* describe how to perform the operation associated with a specific message, providing the actual implementation. When a message is sent to an object, the sender does not know exactly which method will be invoked and how it will execute. This resolution is performed at run-time.

An *interface* is the set of messages to which an object can respond. The sender should know the receiver's interface and can therefore make assumptions about which messages will be accepted. Different objects may have common interfaces with totally different method implementations. For example, integers and floats both respond to the message requesting negation. Nevertheless, the internal implementations of the respective operations are probably quite different.

An object's variables are private and therefore only visible within the object's operations, i.e. the variables can only be manipulated through appropriate methods. This separates the interface from internal implementation details. Thus, objects do not depend on internal details (variables and variable names) of other objects, only on the accepted messages (the object interface).

A.1.2 Classes and Instances

In *Susi* a *class* describes a set of objects with common behavior. These objects described by the class are called *instances* of the class. A class defines the instance methods and variables of its instances. All instances of a common class have the same number (and names) of variables and the same message interface. Every instance has its own private set of these variables in a separate memory region. For example, the class `Integer` describes the behavior of integer objects. Thus, the object `1` is an instance of `Integer`. The class `Integer` specifies which messages are accepted by integers and how they are implemented.

In Smalltalk-80, classes themselves are objects with *class variables* and *class methods*. Since every object is also an instance, every class has a corresponding *metaclass*. The class is an instance of the appropriate metaclass which defines the classes variables and methods, just like the class itself defines the variables and methods of its instances. For example, date objects are described by the class `Date`. The class `Date` itself has a set of class methods (`today`, etc.) described in the metaclass `Date class`. Thus, every class has exactly one metaclass. In *Susi* (like in Smalltalk-80) classes are also objects with class variables and methods. For reasons of simplicity in *Susi* there is only one single metaclass for all classes. Every class is an instance of `MetaClass` which is an instance of itself, introducing a point of circularity in the class-instance relationship. Since there is no separate metaclass for each class, the classes describe their class variables and methods themselves. Although in *Susi* there is an explicit differentiation between classes and their instances, both are treated uniformly as objects.

Susi is based on a number of *core classes* (`Integer`, `Float`, `Null`, etc.) which provide the basic functionality of the programming language like arithmetics and control structures. These are integrated in the system kernel and will be described in greater detail in the following sections. All *Susi* classes are part of a global *class hierarchy*. The class `Object` is the root of the tree and the core classes `Integer`, `True`, `Null`, `Array`, `MetaClass`, etc. are *subclasses* of `Object` (`Object` is their *superclass*). Subclasses *inherit* the object implementation (they are *derived*) from their superclass. `Object` defines the most basic object behavior, the subclasses inherit this behavior and extend or change it appropriately. `Object` itself (being the root of the tree) has no superclass. Every *Susi* class is either directly derived from `Object` or from another existing class; every class is ultimately a subclass of `Object`. Like in Smalltalk-80, only *single inheritance* is supported: every class (except `Object`) has *exactly* one superclass.

A.2 Expressions

Susi source code is represented by sequences of statements and expressions. The statements are evaluated sequentially, each expression returning a value. There are five expression types in *Susi*:

1. *Literals* describe constant objects.
2. *Class names* describe references to global classes.
3. *Variable names* describe references to currently accessible variables.
4. *Messages* describe messages to objects.
5. *Statement blocks* describe objects representing deferred sequences of statements.

A.2.1 Literals

The value of a literal expression is always the same object. The object is created once at compile-time and the literal expression corresponds to a reference to that object. In *Susi*, three objects can be described by literals.

1. `Integer` objects represent whole numbers. An integer literal consists of a sequence of digits and may be preceded by a minus sign. The maximum number of representable integers is determined by a system word, and is therefore system dependent.
2. `Float` objects represent real numbers. A float literal consists of two integer literals separated by a decimal point and may be preceded by a minus sign.
3. `String` objects represent sequences of characters. A string literal is a sequence of ASCII characters delimited by double quotes. Any character (newline, tabspace, etc.) may be included in the string. If a double quote is to be included, it must be duplicated. String literals have a maximum length of 30000 characters. `String` objects are not limited (except by the amount of available memory).

A.2.2 Class Names

The classes defined in *Susi* are globally accessible objects and can be referenced by their names. The class name expression evaluates to the object representing the named class. A class name is a simple identifier except that the initial character *must* be an uppercase letter. This naming convention is enforced by the compiler. For example: `Integer`, `Null`, and `True`.

A.2.3 Variable Names

Variable names are used to reference the currently accessible variables. This makes evident that variable names are statically scoped. They are only legal if the named variables are accessible or visible in the current context. For example, the private variables of an object are only visible within the object's methods. The variable name expression evaluates to the object currently referenced by the named variable. A variable name is a simple identifier with a lowercase initial character. This naming convention is also enforced by the compiler. For example: `myVariable`.

Assignment Prefix

Variables in *Susi* are not constrained by types and can reference instances of any classes. Variables are initialized to `Null` and may then reference any object. Thus, variables in *Susi* are similar to pointers in C/C++. Variables can be assigned to reference objects with the assignment operator (`=>`):

```
myVariable=>"first a string, then a float..."
myVariable=>9.99
```

The assignment operator is an expression prefix (`<variable-name> => expression`) and can be included more than once. Such stacked assignments are evaluated in right-to-left order:

```
variable1=>variable2=>variable3=>0  // initialize all to zero
```

A.2.4 Messages

Objects send messages to other objects, requesting them to perform some sort of operation. In *Susi* practically everything can be accomplished through passing messages. Even the most basic operations such as object comparison, integer arithmetics, boolean operations and control structures can be implemented as messages. Every message consists of three parts: a receiver expression, a message name and possibly some arguments. The receiver expression is evaluated first and returns the message receiver. The receiver expression and the message name are separated by a dot. The message name is followed by the arguments in brackets. If the message has no arguments the empty brackets may be omitted for clarity.

```
Directory.make("/root/tmp/plok")
File.open("/root/tmp/plok/test.susi", "rt")
Date.today
"Hello World!".write
```

On receiving a message, the receiver must decide whether the message can be responded to and whether the receiver understands the message. An association must be made between the message and the receiver's operation implementations (the methods). Methods are described in Section A.3.3. If the corresponding implementation is found then the method can be executed and the result is returned as the result of the message expression[1]. Since messages are normal expressions, they can be *cascaded*. This means that the result of one message expression can directly be taken as the receiver of the next message, and so on: `pictureContainer.get("Ferrari.jpg").gaussianBlur(3).display`.

To precisely specify methods and where they are implemented, we introduce a new notation: a *method signature* is defined as the method name and the arity separated by a colon.

[1] As opposed to C/C++ [Str97], in *Susi* the message arguments are evaluated in left-to-right order. Further, the arguments are evaluated *before* the message is sent—they are evaluated even if the receiver does not understand the message.

For example, the method `add` with one parameter has the signature `add:1`. The object that implements the method is specified by the object name separated from the method signature by ⇒. For example, `Integer ⇒ add:1` refers to integer addition. Class methods are specified by adding `class` to the object name. For example, `Date class ⇒ today:0` refers to the specific class method.

A.2.5 Statement Blocks and Block Objects

Statement blocks describe objects representing deferred sequences of instructions, meaning sequences of statements which can be executed at a later time. A statement block is a sequence of expressions, separated by semicolons and delimited by braces. The semicolon after the last statement in the block can be omitted. For example:

```
{x => x.add(1); x.write}
```

When a statement block expression is encountered, the statements delimited by the braces are not executed immediately. Instead, a new object is created which can execute the enclosed statements at a later time. Such `Block` objects can be treated like normal objects. They can be assigned to variables, passed as message arguments or receive messages themselves. The following statement does not execute the block statements, but assigns the new `Block` object to the variable `nukeWindows`.

```
nukeWin95=>{Win95.prepareToDie; Timer.count(1000); Win95.nuke}
```

`Block` objects can be evaluated by sending them the message `evaluate:0`. For example, `nukeWin95.evaluate`. Like all expressions, the message `evaluate:0` must return a value. The result of the `evaluate:0` message is the value of the last expression in the statement sequence. If the sequence is empty, i.e. `{}`, then the result is `Null`. The following expression evaluates to 2:

```
{x => 0; x => x.add(1); x.add(1)}.evaluate
```

Statement blocks can also define local variables, meaning variables which will be visible only within the block and are part of the `Block` object:

```
{[ temp ] temp=>10}
```

Control Structures

With the help of `Block` objects, both *conditional branches* and *conditional loops* can be implemented with messages. Conditional branches correspond to *if-then-else* statements and conditional loops correspond to *while* statements in other conventional programming languages. In *Susi* the classes `True` and `False` represent the boolean values *true* and *false* and are used as branch/loop conditions. The conditional branch executes a sequence of instructions depending on the result of an expression. For example:

```
(x>0).ifTrue({x.write})
```

This evaluates the expression and sends the resulting object the message `ifTrue` with the `Block` object as an argument. Depending on `x`, the expression may evaluate to `True` or `False`. Both these classes have the class method `ifTrue:1` implemented. `True class` $\Rightarrow$ `ifTrue:1` evaluates the `Block` argument and returns the result, `False class` $\Rightarrow$ `ifTrue:1` doesn't.

Similarly, the boolean classes have the method `ifTrueFalse:2` implemented, which takes two `Block` objects as arguments. `True class` $\Rightarrow$ `ifTrueFalse:2` evaluates the first argument, `False` $\Rightarrow$ `ifTrueFalse:2` evaluates the second argument:

```
(x>10).ifTrueFalse({"greater 10"}, {"not greater 10"}).write
```

The conditional loop executes a sequence of instructions as long as a condition (represented by a statement block) evaluates to `True`:

```
{x>0}.whileTrue({x=>x-1})
```

The receiver of `whileTrue:1` is a `Block` object. The receiver evaluates itself (it sends itself the message `evaluate:0`). If the result is `True`, it evaluates the argument. The argument is repeatedly evaluated as long as the receiver evaluates to `True`.

A.3 Classes and Instances

A.3.1 Class Definitions

Class definitions are used in *Susi* to specify the class and instance behavior. The class definition specifies the class name, the superclass, the class variables, the class methods, the instance variables and the instance methods. For example, the class `Link` is used for heterogeneous linked lists:

```
class Link extends Object
{
    method new {self.error("does not implement new:0")}
    method new (object, next) {
        return super.new.object(object).next(next)}
instance
    [ object next ]
    method object {return object}
    method object(anObject) {object => anObject}
    method next {return next}
    method next(aLink) {next => aLink}
}
```

The keyword `class` is followed by the class name. Class names are identifiers with uppercase initial letters. Next, the keyword `extends` is followed by the name of the superclass. Every class (except `Object`) must have a superclass. The concept of inheritance is elaborated in Section A.4. The following definitions are delimited by braces. The class `Link` does not have any class variables, which would usually be listed first. After the class variables the class methods are defined—their implementation is given. Methods will be explained in Section A.3.3). The keyword `instance` indicates that the class part is over and the following definitions concern the instances. The instance variables are listed in square brackets. Finally, the instance methods are defined.

A.3.2 Instance Creation

New objects can be created by sending messages to classes. The method `new:0` is implemented in `Object class` and is inherited by all the other classes. Upon receiving this message, most classes create a new instance of themselves. For example,

```
class MyClass extends Object {instance [ x y z ] }
[...]
MyClass.new
```

creates a new object as an instance of `MyClass` with the variables `x`, `y`, and `z`. The instance has the private variables and the interface specified in the instance part of the `MyClass` class definition.

Classes can also create new instances when they receive messages other than `new:0`. A common practice is even to forbid instance creation with `new:0` and to force the sender to initialize the instance variables appropriately. An example of this has already been shown:

```
method new {self.error("does not implement new:0")}
method new(object, next) {
    return super.new.object(object).next(next)}
```

These are the class methods of the class `Link`. The method `new:0` is redefined to report an error, because the instance variables cannot be initialized properly without additional information. The method `new:2` is defined, which creates a new instance (using `Object class` ⇒ `new:0`) and initializes the variables. The pseudo-variable `self` is explained in sections A.3.3 and A.4.3. The pseudo-variable `super` will be explained in Section A.4.4. For example, the following code fragment creates a new linked list which holds the objects `1` and `"last element"` (the list is terminated by `Null`):

```
root => Link.new(1, Link.new("last element", Null))
```

A.3.3 Methods

Methods describe how to perform the operation requested by a specific message. They provide the actual operation implementation. For example:

```
method sum(x, y) {return x.add(y)}
method pi {return 3.1415}
```

The keyword `method` is followed by the method name. Method names are identifiers with lowercase initial letters. The method parameters are specified in brackets and are separated by commas. If the method has no parameters, the empty brackets may be omitted. Finally, the method body is specified in the form of a statement block. When an object receives a message, it searches for an appropriate implementation. If a corresponding method is found, it is executed. Otherwise an error is issued, indicating that the object does not understand the message.

Local Variables

The method body (being a statement block) can define local variables. These are named temporary variables only visible and accessible from within the method body. Actually, the method parameters themselves are converted to local block variables assigned to the arguments when the method is invoked. Local variables are otherwise initialized to `Null`. For example, the following method could be defined in `Array` and be used to swap two array elements.

```
method swap(index1, index2)
{
    [ temp ]
    temp => self.index(index1);
    self.indexPut(index1, self.index(index2));
    self.indexPut(index2, temp)
}
```

Variable names must be unique within a scope. A variable name may not be used for a new variable if a variable with the same name is already visible in the current scope. For example, the following class definition is illegal, since the variable `x` is used as an object variable and as a parameter:

```
class X extends Y {[ x ] method test(x) {}}
```

Return Values

Message expressions always evaluate to an object. Therefore, the methods which implement the operations must specify a return value. *Susi* offers the classic *return statement* to abort the method execution and give back a return value to the message sender.

```
method zero {return 0}
```

If the method body doesn't contain a return statement, when the method terminates the receiver of the message is returned as the message expression result. For example, the following method returns the receiver.

```
method doNothing {}
```

The return statement aborts the control of the *current* method. If a block with a return statement is evaluated, the current method is terminated, no matter where the block is from. This means that if a method passes a block with a return statement to another method and the other method evaluates the block, then only the other method is aborted, not the method that created the block. This should always be kept in mind when passing blocks with return statements as arguments.

Self

In the class methods of `Link`, the method `new:0` contained the pseudo-variable `self`. This pseudo-variable cannot be changed and refers to the receiver of the message. By using `self`, an object can send messages to itself, or pass itself as an argument. The prior example causes the object to send itself a message requesting to report an error. A common use of `self` is to implement recursion. For example:

```
method hanoi(number, from, to, over)
{
    (number>0).ifTrue({
        self.hanoi(number-1, from, over, to);
        self.hanoi(number-1, over, to, from)})
}
```

The variable `self` always evaluates to the receiver of the method where `self` occurs. For example: `object1` executes a method containing a statement block which includes a `self` reference. This `self` reference will always evaluate to `object1`, no matter where the `Block` object is actually executed. So, if the statement block is evaluated and the resulting `Block` object is passed as an argument to `object2`, when `object2` sends `evaluate:0` to the `Block` object, the `self` reference in the block will still evaluate to `object1`. Consider the following classes:

```
class X extends Object {
    method test {Y.test({self})}
}
class Y extends Object {
    [ a ]
    method test(block) {a => block.evaluate}
}
```

`X class` ⇒ `test:0` creates a `Block` object containing a `self` reference. The `Block` object is passed as an argument to `Y class` ⇒ `test:1`, which evaluates the object by sending `evaluate:0` and assigns the result to the class variable `a`. Although the block is evaluated by `Y`, the `self` reference still evaluates to `X`. Therefore, after executing the expression `X.test`, the class variable `a` refers to `X`.

Overloading

Not only the method name but also the arity is decisive for identifying a method. Thus, methods can be *overloaded on arity*. This means that a method name can be used for different methods as long as the number of parameters is different. For example, the methods `sum:2`, `sum:3`, and `sum:4` refer to totally different and unrelated methods:

```
method sum(x, y) {return x.add(y)}
method sum(x, y, z) {return x.add(y).add(z)}
method sum(a, b, c, d) {return a.add(b).add(c).add(d)}
```

A.3.4 Class Extensions

Usually, methods are defined as part of the class definition. This is not always possible. For example, all the core classes are integrated in *Susi* and defined implicitly. They don't have class definitions as source code and thus cannot be extended. *Susi* offers a second mechanism for defining methods. A *class extension* is defined outside of the class definition and specifies a method and which class should be extended. For example:

```
method ifTrue(block) {
    return block.evaluate} extends True class;
method ifTrue(block) {
    } extends False class;
```

The method definition is followed by the keyword `extends`. Then, the destination class name is specified and whether the method should be added as a `class` or `instance` method. The following example illustrates an instance extension:

```
method rounded {
    return (self.add(0.5)).asInteger} extends Float instance;
```

A.4 Inheritance

A.4.1 Class Hierarchy

Every class is part of the *Susi class hierarchy*. The core classes form the basic class hierarchy which is shown in figure A.1. Black arrows denote subclass relations; gray arrows denote instance relations.

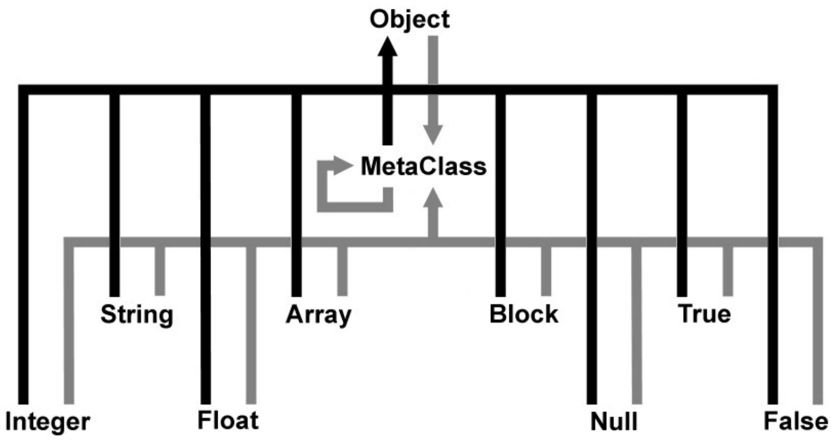

Figure A.1: Core Class Hierarchy

The class hierarchy can be extended by adding user-defined classes. The class `Object` defines the most basic object behavior and is the root of the tree. *Susi* supports single inheritance: every class (except `Object`) has *exactly* one superclass. Since classes are normal objects, they themselves must be instances of classes. For reasons of simplicity, in *Susi* there is only one single metaclass for all classes. Every class is an instance of `MetaClass`, which is an instance of itself.

A.4.2 Extending and Changing Implementations

A subclass inherits the implementation from the superclass and can extend or change it appropriately. The subclass inherits the class variables, the class methods, the instance variables and the instance methods. New variables can be added, but inherited variables may not be redeclared.

```
class X extends Object {[ x ]}
class Y extends X {[ y ]}  // inherits class variable x
```

The functionality of the objects can be extended by adding new methods. These new methods are available only to the instances of the subclass. Thus, the interface of the subclass instances is extended. The functionality of the objects can be changed by *overriding* inherited methods. The interface stays the same but the subclass can redefine methods, thus reacting differently to messages than the superclass. Assuming the class definitions

```
class X extends Object {
instance
    method test {return 0}
}
class Y extends X {
instance
    method test {return 1}  // override test:0
}
```

the message `X.new.test` (create a new instance of `X` and send it the message `test:0`) evaluates to `zero` while the message `Y.new.test` evaluates to `one` because `Y` overrides `test:0`.

A.4.3 Messages to Self

The pseudo-variable `self` evaluates to the receiver of the message, independent of where in the class hierarchy the current method is defined. Consider the following classes:

```
class X extends Object {
instance
    method plok {return 0}
    method getPlok {return self.plok}
}
class Y extends X {
instance
    method plok {return 1}
}
```

The class `X` defines `plok:0` and `getPlok:0`. The second method causes the receiver to send itself the message `plok:0`. The class `Y` overrides `plok:0` to return a different value. The expression `X.new.getPlok` evaluates to `zero`, while `Y.new.getPlok` evaluates to `one`, because even though `getPlok:0` was defined as an instance method of `X`, the `self` reference resolves to the instance of `Y`.

A.4.4 Messages to Super

When introducing class definitions, the example class `Link` defined a method which contained the pseudo-variable `super`. Messages to `super` are similar to messages to `self`. Both pseudo-variables refer to the receiver. The difference is as follows: when a message is sent to `super`, the search for the appropriate method doesn't start in the receiver's class but in the superclass of the class *defining* the method.

Messages to `self` start the method search in the class of the receiver, no matter where in the class hierarchy the current method is defined. Messages to `super` also send the message to the receiver, but the search for the appropriate method starts in the superclass of the class defining the current method (and not in the superclass of the receiver's class).

Returning to the previous example, the class `Link` overrides `new:0` to report an error. The class method `new:2` calls the superclass method `Object class` ⇒ `new:0` in order to create a new instance before setting the variables. No matter how many classes are thenceforth derived from `Link`, whenever `Link class` ⇒ `new:2` is executed the expression `super.new` will always call `Object class` ⇒ `new:0`.

A.5 Garbage Collection

Manual memory management is usually error prone. Code tends to contain memory leaks which (although tolerable in short-running applications) are critical in long-running programs. Automatic storage reclamation frees the programmer from the burden of explicit memory deallocation, thus eliminating this common source of errors. The availability of automatic storage reclamation in a programming language can boost a programmer's productivity, eliminate memory management errors by closing memory leaks and encourage component reuse by facilitating modular programming.

Garbage collection has a great impact on object-oriented programming, greatly simplifying the interfaces between interacting components. The necessity for exposing memory management details (who is responsible for deallocating shared objects) becomes obsolete. Thus, modular programming and code reuse are enforced. Without garbage collection, supposedly independent modules are forced to cooperate more closely than actually required—the object-oriented principles encapsulation and implementation-hiding are broken.

Susi supports automatic storage reclamation. The runtime system determines what memory previously allocated by the program is no longer being used. These memory regions are "freed" for further reallocation. *Susi* uses a simple, non-generational mark-and-sweep algorithm to perform garbage collection.

Bibliography

[AC93] Roberto Amadio, Luca Cardelli. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15(4):575–631, 1993.

[Ada83] American National Standards Institute. *Reference Manual for the Ada Programming Language Ansi/Mil-Std-1815A-1983*. Springer-Verlag, 1983.

[ADL91] Rakesh Agrawal, Linda DeMichiel, Bruce Lindsay. Static Type Checking of Multi-Methods. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 113–128, Phoenix, October 1991. Published as *ACM SIGPLAN Notices* 26(11), November 1991.

[AFM97] Ole Agesen, Stephen Freund, John Mitchell. Adding Type Parameterization to the Java Language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, October 1997. Published as *ACM SIGPLAN Notices* 32(10), October 1997.

[AGS91] H. Al-Haddad, K. George, M. Samadzadeh. Approaches to Reusability in C++ and Eiffel. *Journal of Object-Oriented Programming* 4(5):34–45, September 1991.

[AL90] Pierre America, Frank van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *Proceedings of the European Conference on Object-Oriented Programming/Conference on Object-Oriented Programming, Systems, Languages, and Applications (ECOOP/OOPSLA)*, pages 161–168, Ottawa, October 1990. Published as *ACM SIGPLAN Notices* 25(10), October 1990.

[Ame87] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Paris, June 1987. Published as *Lecture Notes in Computer Science (LNCS)* 276:234–242, Springer-Verlag, 1987.

[Ame90] Pierre America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, Noordwijkerhout, The Netherlands, May/June 1990. Published as *Lecture Notes in Computer Science (LNCS)* 489:60–90, Springer-Verlag, 1991.

[ASB94] S. Alagić, R. Sunderraman, R. Bagai. Declarative Object-Oriented Programming: Inheritance, Subtyping and Prototyping. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Bologna, July 1994. Published as *Lecture Notes in Computer Science (LNCS)* 812:236–259, Springer-Verlag, 1994.

[BC90] Gilad Bracha, William Cook. Mixin-based Inheritance. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 303–311, Ottawa, October 1990. Published as *ACM SIGPLAN Notices* 25(10), October 1990.

[BC96] John Boyland, Giuseppe Castagna. Type-Safe Compilation of Covariant Specialization: A Practical Case. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Linz, July 1996. Published as *Lecture Notes in Computer Science (LNCS)* 1098:3–25, Springer-Verlag, 1996.

[BC97] John Boyland, Giuseppe Castagna. Parasitic Methods: An Implementation of Multi-Methods for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 66–76, Atlanta, October 1997. Published as *ACM SIGPLAN Notices* 32(10), October 1997.

[BCC+96] Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary Leavens, Benjamin Pierce. On Binary Methods. *Theory and Practice of Object Systems* 1(3):221–242, 1996.

[BG96] Myles Barrett, Marshall Giguere. A Note on Covariance and Contravariance Unification. *ACM SIGPLAN Notices* 31(1):32–35, January 1996.

[BGM89] Gordon Blair, John Gallagher, Javad Malik. Genericity vs. Inheritance vs. Delegation vs. Conformance vs.... *Journal of Object-Oriented Programming* 2(3):11–17, September/October 1989.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy. Object Structure in the Emerald System. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 78–86, Portland, September/October 1986. Published as *ACM SIGPLAN Notices* 21(11), November 1986.

[BI82] Alan Borning, Daniel Ingalls. Multiple Inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 234–237, Pittsburgh, August 1982.

[BI94] Kenneth Baclawski, Bipin Indurkhya. The Notion of Inheritance in Object-Oriented Programming. *Communications of the ACM (CACM)* 37(9):118–119, September 1994.

[BKK+86] Daniel Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 17–29, Portland, September/October 1986. Published as *ACM SIGPLAN Notices* 21(11), November 1986.

[BOW98] Kim Bruce, Martin Odersky, Philip Wadler. A Statically Safe Alternative to Virtual Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Brussels, July 1998. Published as *Lecture Notes in Computer Science (LNCS)* 1445:523–549, Springer-Verlag, 1998.

[BP93] Andrew Black, Jens Palsberg. Foundations of Object-Oriented Languages: Workshop Report. In *Workshop on Foundations of Object-Oriented Languages*, California, October 1993. Published as *ACM SIGPLAN Notices* 29(3):3–11, March 1994.

[BPS89] Günther Blaschek, Gustav Pomberger, Alois Stritzinger. A Comparison of Object-Oriented Programming Languages. *Structured Programming* 10(4):187–197, 1989.

[BR97] Gerald Baumgartner, Vincent Russo. Implementing Signatures for C++. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19(1):153–187, January 1997.

[Bro98] Benjamin Brosgol. A Comparison of Ada and Java as a Foundation Teaching Language. *Ada Letters* 18(5):12–38, September/October 1998.

[Budd91] Timothy Budd. Generalized Arithmetic in C++. *Journal of Object-Oriented Programming* 3(6):11–22, February 1991.

[BW86] Kim Bruce, Peter Wegner. An Algebraic Model of Subtypes in Object-Oriented Languages (Draft). *ACM SIGPLAN Notices* 21(10):163–172, October 1986.

[Car88] Luca Cardelli. A Semantics of Multiple Inheritance. *Information and Computation* 76:138–164, 1988.

[Car88b] Luca Cardelli. Structural Subtyping and the Notion of PowerType. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 70–79, San Diego, January 1988.

[Car91] Luca Cardelli. Typeful Programming. In *Formal Description of Programming Concepts*, Springer-Verlag, 1991.

[Cas95] Giuseppe Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17(3):431–447, May 1995.

[CC99] Craig Chambers, Weinin Chen. Efficient Multiple and Predicate Dispatching. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 238–255, Denver, November 1999. Published as *ACM SIGPLAN Notices* 34(10), October 1999.

[CCHO89] Peter Canning, William Cook, Walter Hill, Walter Olthoff. Interfaces for Strongly-Typed Object-Oriented Programming. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 457–467, New Orleans, October 1989. Published as *ACM SIGPLAN Notices* 24(10), October 1989.

[CG90] Bernard Carré, Jean-Marc Geib. The Point of View Notion for Multiple Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming/Conference on Object-Oriented Programming, Systems, Languages, and Applications (ECOOP/OOPSLA)*, pages 312–321, Ottawa, October 1990. Published as *ACM SIGPLAN Notices* 25(10), October 1990.

[Cha92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Utrecht, June/July 1992. Published as *Lecture Notes in Computer Science (LNCS)* 615:33–56, Springer-Verlag, 1992.

[Cha93] Craig Chambers. Predicate Classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Kaiserslautern, July 1993. Published as *Lecture Notes in Computer Science (LNCS)* 707:268–296, Springer-Verlag, 1993.

[CGL95] Giuseppe Castagna, Giorgio Ghelli, Giuseppe Longo. A Calculus for Overloaded Functions with Subtyping. *Information and Computation* 117:115–135, 1995.

[CHC90] William Cook, Walter Hill, Peter Canning. Inheritance Is Not Subtyping. In *Conference Record of the 17^{th} Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 125–135, San Francisco, January 1990.

[CL94a] Giuseppe Castagna, Gary Leavens. Foundations of Object-Oriented Languages: 2^{nd} Workshop Report. In *Workshop on Foundations of Object-Oriented Languages*, Paris, July 1994. Published as *ACM SIGPLAN Notices* 30(2):5–11, February 1995.

[CL94b] Craig Chambers, Gary Leavens. Typechecking and Modules for Multi-Methods. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–15, Portland, October 1994. Published as *ACM SIGPLAN Notices* 29(10), October 1994.

[CL95] Jan-Bon Chen, Samuel Lee. Pursuing Safe Polymorphism in OOP. *Journal of Object-Oriented Programming* 8(1):39–45, March/April 1995.

[Cook87a] Steve Cook. OOPSLA'87 Panel P2: Varieties of Inheritance. In *Addendum to the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, October 1987. Published as *ACM SIGPLAN Notices* 23(5):35–40, May 1988.

[Cook90] William Cook. Object-Oriented Programming Versus Abstract Data Types. In *Foundations of Object-Oriented Languages, REX School/Workshop*, Noordwijkerhout, The Netherlands, May/June 1990. Published as *Lecture Notes in Computer Science (LNCS)* 489:151–178, Springer-Verlag, 1991.

[Cook92] William Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–15, Vancouver, October 1992. Published as *ACM SIGPLAN Notices* 27(10), October 1992.

[CP89] William Cook, Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 433–443, New Orleans, October 1989. Published as *ACM SIGPLAN Notices* 24(10), October 1989.

[CRR98] Lobel Crnogorac, Anand Rao, Kotagiri Ramamohanarao. Classifying Inheritance Mechanisms in Concurrent Object-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Brussels, July 1998. Published as *Lecture Notes in Computer Science (LNCS)* 1445:517–600, Springer-Verlag, 1998.

[Cus91] Elspeth Cusack. Inheritance in Object Oriented Z. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Geneva, July 1991. Published as *Lecture Notes in Computer Science (LNCS)* 512:167–179, Springer-Verlag, 1991.

[CW85] Luca Cardelli, Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys* 17(4):471–522, December 1985.

[Dav92] Stephen Davis. C++ Objects that Change their Types. *Journal of Object-Oriented Programming* 5(4):27–32, July/August 1992.

[DD85] James Donahue, Alan Demers. Data Types are Values. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7(3):426–445, July 1985.

[DE97] Sophia Drossopoulou, Susan Eisenbach. Java is Type Safe — Probably. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä, June 1997. Published as *Lecture Notes in Computer Science (LNCS)* 1241:389–418, Springer-Verlag, 1997.

[DGLM95] Mark Day, Robert Gruber, Barbara Liskov, Andrew Myers. Subtypes vs. Where Clauses: Constraining Parametric Polymorphism. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 156–168, Austin, October 1995. Published as *ACM SIGPLAN Notices* 30(10), October 1995.

[DN66] O. Dahl, K. Nygaard. Simula – An Algol-based Simulation Language. *Communications of the ACM (CACM)* 9(9):671–678, September 1966.

[DT88] Scott Danforth, Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys* 20(1):29–72, March 1988.

[DT92] Mahesh Dodani, Chung-Shin Tsai. ACTS: A Type System for Object-Oriented Programming Based on Abstract and Concrete Classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Utrecht, June/July 1992. Published as *Lecture Notes in Computer Science (LNCS)* 615:309–328, Springer-Verlag, 1992.

[Eck95] Bruce Eckel. *Thinking in C++*. Prentice Hall, 1995.

[Frø92] Svend Frølund. Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Utrecht, June/July 1992. Published as *Lecture Notes in Computer Science (LNCS)* 615:185–196, Springer-Verlag, 1992.

[Ghe91] Giorgio Ghelli. A Static Type System for Message Passing. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 129–145, Phoenix, October 1991. Published as *ACM SIGPLAN Notices* 26(11), November 1991.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GJS96] James Gosling, Bill Joy, Guy Steele. *The Java^{TM} Language Specification.* Addison-Wesley, 1996.

[GR83] Adele Goldberg, David Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.

[Har91] Warren Harris. Contravariance for the rest of us. *Journal of Object-Oriented Programming* 4(7):10–18, November/December 1991.

[Hau93] Franz Hauck. Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 231–239, Washington, September/October 1993. Published as *ACM SIGPLAN Notices* 28(10), October 1993.

[HB87] Daniel Halbert, Patrick O'Brien. Using Types and Inheritance in Object-Oriented Languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Paris, June 1987. Published as *Lecture Notes in Computer Science (LNCS)* 276:20–31, Springer-Verlag, 1987.

[Hen86] James Hendler. Enhancement for Multiple-Inheritance. *ACM SIGPLAN Notices* 21(10):98–106, October 1986.

[Höl93] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Kaiserslautern, July 1993. Published as *Lecture Notes in Computer Science (LNCS)* 707:36–56, Springer-Verlag, 1993.

[Horn87] Chris Horn. Conformance, Genericity, Inheritance and Enhancement. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Paris, June 1987. Published as *Lecture Notes in Computer Science (LNCS)* 276:223–233, Springer-Verlag, 1987.

[Hür94] Walter Hürsch. Should Superclasses be Abstract?. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Bologna, July 1994. Published as *Lecture Notes in Computer Science (LNCS)* 812:12–31, Springer-Verlag, 1994.

[Ing86] Daniel Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 347–349, Portland, September/October 1986. Published as *ACM SIGPLAN Notices* 21(11), November 1986.

[JA92] Suresh Jagannathan, Gul Agha. A Reflective Model of Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Utrecht, June/July 1992. Published as *Lecture Notes in Computer Science (LNCS)* 615:350–371, Springer-Verlag, 1992.

[Jon92] Rick Jones. Extended Type Checking in Eiffel. *Journal of Object-Oriented Programming* 5(2):59–62, May 1992.

[JZ91] Ralph Johnson, Jonathan Zweig. Delegation in C++. *Journal of Object-Oriented Programming* 4(7):31–34, November/December 1991.

[Kee89] S. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989.

[Kil92] Michael Kilian. Trellis: What We Have Learned from a Strongly Typed Language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 91, Vancouver, October 1992. Published as *ACM SIGPLAN Notices* 27(10), October 1992.

[KLMM94] Dinesh Katiyar, David Luckham, John Mitchell, Sigurd Meldal. Polymorphism and Subtyping in Interfaces. In *Proceedings of the Workshop on Interface Definition Languages*, pages 22–34, Portland, January 1994. Published as *ACM SIGPLAN Notices* 29(8), August 1994.

[KMN+90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *Journal of Object-Oriented Programming* 3(3):11–22, September/October 1990.

[Kur99] Stoyan Kurtev. *Susi (v3.1), An Object-Oriented Programming Language, User's Guide*. Technical Report, February 1999. Available on the WWW via the URL `http://www.complang.tuwien.ac.at/susi/Manuals.zip`.

[Lam93] John Lamping. Typing the Specialization Interface. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 201–214, Washington, September/October 1993. Published as *ACM SIGPLAN Notices* 28(10), October 1993.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 214–223, Portland, September/October 1986. Published as *ACM SIGPLAN Notices* 21(11), November 1986.

[Lis87] Barbara Liskov. Data Abstraction and Hierarchy. In *Addendum to the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Orlando, October 1987. Published as *ACM SIGPLAN Notices* 23(5):17–34, May 1988.

[Liu91] Chung-Shyan Liu. On the Object-Orientedness of C++. *ACM SIGPLAN Notices* 26(3):63–67, March 1991.

[LP91] Wilf LaLonde, John Pugh. Subclassing ≠ Subtyping ≠ Is-a. *Journal of Object-Oriented Programming* 3(5):57–62, January 1991.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, Craig Schaffert. Abstraction Mechanisms in CLU. *Communications of the ACM (CACM)* 20(8):564–576, August 1977.

[LT93] H. Lee, W. Tsai. A New Partial Inheritance Mechanism and its Applications. *Journal of Object-Oriented Programming* 6(4):53–63, July/August 1993.

[LW90] Gary Leavens, William Weihl. Reasoning about Object-Oriented Programs that Use Subtypes (Extended Abstract). In *Proceedings of the European Conference on Object-Oriented Programming/Conference on Object-Oriented Programming, Systems, Languages, and Applications (ECOOP/OOPSLA)*, pages 212–223, Ottawa, October 1990. Published as *ACM SIGPLAN Notices* 25(10), October 1990.

[LW93] Barbara Liskov, Jeannette Wing. A New Definition of the Subtype Relation. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Kaiserslautern, July 1993. Published as *Lecture Notes in Computer Science (LNCS)* 707:118–141, Springer-Verlag, 1993.

[LW94] Barbara Liskov, Jeannette Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16(6):1811–1841, November 1994.

[LZ74] Barbara Liskov, Stephen Zilles. Programming with Abstract Data Types. In *Proceedings of a Symposium on Very High Level Languages*, pages 50–59, Santa Monica, March 1974. Published as *ACM SIGPLAN Notices* 9(4), April 1974.

[MDCB91] R. Morrison, A. Dearle, R. Connor, A. Brown. An Ad-Hoc Approach to the Implementation of Polymorphism. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13(3):342–371, 1991.

[Mey88] Bertrand Meyer. Genericity Versus Inheritance. *Journal of Pascal, Ada & Modula-2* 7(2):13–30, March/April 1988.

[Mey90] Bertrand Meyer. Lessons from the Design of the Eiffel Libraries. *Communications of the ACM (CACM)* 33(9):68–88, September 1990.

[Mey92] Bertrand Meyer. Ensuring Strong Typing in an Object-Oriented Language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 89–90, Vancouver, October 1992. Published as *ACM SIGPLAN Notices* 27(10), October 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[MHH91] Warwick Mugridge, John Hamer, John Hosking. Multi-Methods in a Statically-Typed Programming Language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Geneva, July 1991. Published as *Lecture Notes in Computer Science (LNCS)* 512:307–324, Springer-Verlag, 1991.

[MMM90] Ole Madsen, Boris Magnusson, Birger Møller-Pedersen. Strong Typing of Object-Oriented Languages Revisited. In *Proceedings of the European Conference on Object-Oriented Programming/Conference on Object-Oriented Programming, Systems, Languages, and Applications (ECOOP/OOPSLA)*, pages 140–149, Ottawa, October 1990. Published as *ACM SIGPLAN Notices* 25(10), October 1990.

[Moon86] David Moon. Object-Oriented Programming with Flavors. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–8, Portland, September/October 1986. Published as *ACM SIGPLAN Notices* 21(11), November 1986.

[MT97] Robert Miller, Anand Tripathi. Issues with Exception Handling in Object-Oriented Systems. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä, June 1997. Published as *Lecture Notes in Computer Science (LNCS)* 1241:85–103, Springer-Verlag, 1997.

[Ped89] Claus Pedersen. Extending Ordinary Inheritance Schemes to Include Generalization. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 407–417, New Orleans, October 1989. Published as *ACM SIGPLAN Notices* 24(10), October 1989.

[Por92] Harry Porter, III. Separating the Subtype Hierarchy from the Inheritance of Implementation. *Journal of Object-Oriented Programming* 4(9):20–29, February 1992.

[PS91] Jens Palsberg, Michael Schwartzbach. What is Type-Safe Code Reuse?. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Geneva, July 1991. Published as *Lecture Notes in Computer Science (LNCS)* 512:325–341, Springer-Verlag, 1991.

[PS92] Jens Palsberg, Michael Schwartzbach. Three Discussions in Object-Oriented Typing (Report on ECOOP'91 Workshop W5). *OOPS Messenger* 3(2):31–38, April 1992.

[Pun99] Franz Puntigam. *Typsysteme*. Lecture Notes, October 1999

[RFW96] Jonathan Rossie, Daniel Friedman, Mitchell Wand. Modeling Subobject-Based Inheritance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Linz, July 1996. Published as *Lecture Notes in Computer Science (LNCS)* 1098:248–274, Springer-Verlag, 1996.

[Roy98] Jean-Claude Royer. Type Checking Object-Oriented Programs: Core of the Problem and Some Solutions. *Journal of Object-Oriented Programming* 11(6):58–71, October 1998.

[Rum93] James Rumbaugh. Disinherited! Examples of Misuse of Inheritance. *Journal of Object-Oriented Programming* 5(9):22–24, February 1993.

[Sal98] Peter Salus. *Handbook of Programming Languages, Volume I: Object-Oriented Programming Languages.* Macmillian Technical Publishing, 1998.

[SCB+86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, Carrie Wilpolt. An Introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 9–16, Portland, Oregon, September/October 1986. Published as *ACM SIGPLAN Notices* 21(11), November 1986.

[SCC+93] Yen-Ping Shan, Tom Cargill, Brad Cox, William Cook, Mary Loomis, Alan Snyder. Is Multiple Inheritance Essential to OOP? (Panel). In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 360–363, Washington, September/October 1993. Published as *ACM SIGPLAN Notices* 28(10), October 1993.

[Sei87] Ed Seidewitz. Object-Oriented Programming in Smalltalk and Ada. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 202–213, Orlando, October 1987. Published as *ACM SIGPLAN Notices* 22(12), December 1987.

[Sha94] David Shang. Covariant Specification. *ACM SIGPLAN Notices* 29(12):58–65, December 1994.

[Sin95] Ghan Bir Singh. Single Versus Multiple Inheritance in Object-Oriented Programming. *OOPS Messenger* 6(1):30–39, January 1995.

[SJ94] Krishnamurthy Srinivasan, Sundaresan Jayaraman. A Comparison of Object-Oriented Programming Languages: The Enterprise Modeling Framework Perspective. *Journal of Object-Oriented Programming* 7(3):27–32, June 1994.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 38–45, Portland, Oregon, September/October 1986. Published as *ACM SIGPLAN Notices* 21(11), November 1986.

[Sou97] Desmond D'Souza. Types and Classes: a Language-Independent View. *Journal of Object-Oriented Programming* 10(1):10–13, March-April 1997.

[Str87] Bjarne Stroustrup. What is "Object-Oriented Programming"?. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Paris, June 1987. Published as *Lecture Notes in Computer Science (LNCS)* 276:51–70, Springer-Verlag, 1987.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[Tai96] Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys* 28(3):438–479, September 1996.

[Tho97] Kresten Thorup. Genericity in Java with Virtual Types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä, June 1997. Published as *Lecture Notes in Computer Science (LNCS)* 1241:444–471, Springer-Verlag, 1997.

[TW95] Pete Thomas, Ray Weedon. *Object-Oriented Programming in Eiffel*. Addison-Wesley, 1995.

[US87] David Ungar, Randall Smith. Self: The Power of Simplicity. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 227–242, Orlando, October 1987. Published as *ACM SIGPLAN Notices* 22(12), December 1987.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 168–182, Orlando, October 1987. Published as *ACM SIGPLAN Notices* 22(12), December 1987.

[Weg88] Peter Wegner. Workshop on Object-Oriented Programming (ECOOP87): Types and Inheritance. *ACM SIGPLAN Notices* 23(1):16–21, January 1988.

[Whe97] David Wheeler. Ada, C, C++, and Java vs. The Steelman. *Ada Letters* 17(4):88–112, July/August 1997.

[Win92] Jürgen Winkler. Objectivism: "Class" Considered Harmful. *Communications of the ACM (CACM)* 35(8):128–130, August 1992.

[Wing87] Jeannette Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9(1):1–24, January 1987.

[WZ88] Peter Wegner, Stanley Zdonik. Inheritance as an Incremental Modification Mechanism *or* What Like Is and Isn't Like. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Oslo, August 1988. Published as *Lecture Notes in Computer Science (LNCS)* 322:55–77, Springer-Verlag, 1988.