Compound Types for Java

Martin Büchi and Wolfgang Weck
Turku Centre for Computer Science (TUCS)
Åbo Akademi University
Lemminkäisenkatu 14A, FIN-20520 Turku
Martin.Buechi@abo.fi, Wolfgang.Weck@abo.fi

Abstract

Type compatibility can be defined based on name equivalence, that is, explicit declarations, or on structural matching. We argue that component software has demands for both. For types expressing individual contracts, name equivalence should be used so that references are made to external semantical specifications. For types that are composed of several such contracts, the structure of this composition should decide about compatibility.

We introduce compound types as the mechanism to handle such compositions. To investigate the integrability into a strongly typed language, we add compound types to Java and report on a mechanical soundness proof of the resulting type system.

Java users benefit from the higher expressiveness of the extended type system. We introduce compound types as a strict extension of Java, that is without invalidating existing programs. In addition, our proposal can be implemented on the existing Java Virtual Machine.

1 Introduction

One of several reasons to use Java is its support of component-oriented programming, the creation of compiled building blocks to be used in different contexts, and the assembly of systems from such components. JavaBeans [34], Java's component model, competes with other component software standards, such as CORBA [13] and Microsoft's COM [32], but the language itself may also be used to program to these language independent standards.

Type systems, such as Java's, help to document and safeguard component interfaces. By annotating inter-component call parameters with types, one provides some primitive documentation on how to use a service and at the same time expresses a statically checkable precondition: the object passed must implement certain methods, as stated by the type.

Explicitly declared and named types can stand for contracts about services. The behavioral specification is documented separately and linked to the type via the name. A compiler can, of course, not check compliance with such a specification, but it can verify that references to the same types, and intentionally the same contracts, have been made. Explicitly stated contracts are particularly important in the component software realm [35].

Frequently, classes need to conform to more than one contract. For instance, Microsoft's OLE [6] defines ActiveX control containers via a bundle of contracts to be implemented. Java supports multiple subtyping to this end. Similarly, one may want to declare variables or method parameters of a type comprising several contracts. This is not supported to the same degree by Java. On a first glance it may seem to be no problem because one only would have to declare the right subtype. We will demonstrate, however, that this may not be possible with independently developed software components.

This problem is explained in Sect. 2. In Sect. 3 we take the problem to its root, the question whether type compatibility is being decided by name equivalence or structural match. We will show that we need a mixture of both, and therefore, we propose compound types in Sect. 4.[1] In Sect. 5 we show how to add compound types to Java. They are a strict extension, that is, existing Java programs need not be changed. This also holds for the run-time support, the byte code and the virtual machine in particular. We illustrate the latter in Sect. 6. In Sect. 7 we report on a mechanically verified soundness proof for the extended type system. Section 8 relates to other work and Sect. 9 summarizes our conclusions.

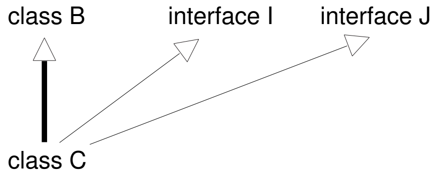
2 The Problem

The problem with Java's type system is explained in this section. In 2.1 we briefly review the relevant aspects of Java's type system. In the second subsection, we introduce the essentials of component software, the domain in which the problem mainly surfaces. With these preliminaries we show that it may be impossible to sharply type a parameter to demand a specific combination of interfaces, so that existing or independently developed classes need not be modified to be compatible.

2.1 Java's type system

An essential ingredient of object-oriented programming and component software is polymorphism. In Java, subtyping relationships can be declared in three ways: a class can subclass another class, a class can implement an interface, and an interface can extend another interface. Subclassing provides for code inheritance in addition to subtyping, whereas interfaces are pure types, that is, no code is attached to them. Typically, multiple subtyping needs less extra conflict resolution rules than multiple subclassing. Thus, Java's designers decided that a class can have only a single direct superclass but implement several interfaces. Because a class without an explicitly declared superclass implicitly inherits from a predefined class `Object`, every class —except `Object`— subtypes exactly one

[1] The compound types defined in this paper are not to be confused with structured types (records, arrays, functions), which are sometimes also called compound types.



class C **extends** B **implements** I, J { ... };

Figure 1: A class extending a base class and implementing two interfaces

class directly and an arbitrary number, possibly zero, of interfaces, as illustrated in Fig. 1.

If a class C is declared to implement an interface I, all methods defined by I exist in C, too. Thus, it is type-safe to assign instances of class C to variables of type I.

We have to take a closer look at the situation in which a class C implements several interfaces, say I and J, that both declare a method with the same name, m. If I and J both declare m with exactly the same signature and return type, C defines m only once and binds this to both interfaces. If the parameter lists differ, Java's overloading mechanism takes care of the situation. Both versions exist within C and upon a call the one with the best fitting signature is selected.[2] The case in which the parameter lists are equal but the return types differ is not permitted by the overloading rules. Consequently, the compiler rejects the declaration of C if I and J conflict in this way.

Subtyping relations can also be established between interfaces. An interface I can be declared to extend other interfaces J0, . . ., Jn. A class implementing I, implicitly implements all interfaces J0, . . ., Jn, but not vice versa. Interfaces cannot subtype (extend) classes.

Whereas subclassing is basically a mechanism for code reuse, multiple subtyping offered via interfaces can be used to express different aspects of objects. For instance, a class D of objects being an applet, runnable in a separate thread, and wanting to be informed of changes in observable objects, would subclass the class Applet and implement the interfaces Runnable and Observer.

These combinations are stated without the need to declare a new type first. The class D automatically constitutes such a new type of its own. Another class, E, also extending Applet and implementing the same interfaces as D, establishes a new type too, but a different one. Java uses name equivalence of types, that is, two types are compatible only if declared so. With classes D and E being declared as separate types, instances of one cannot be assigned to variables of the other's type. Figure 2 summarizes the discussed aspects of Java's type system.

- code inheritance via single subclassing
- multiple subtyping via interfaces (without code inheritance)
- conflict resolution (partially) via overloading
- only types declared to be compatible are compatible (name equivalence)

Figure 2: Relevant aspects of Java's type system

[2] This may not be decidable, in which case an error is flagged, as defined by Java's rules about overloading.

2.2 Component software

One purpose of using object-oriented technology is to create building blocks to be used in several systems. To support such building blocks, Java features, for instance, separate compilation of classes and bundling of related classes as packages.

Component software tries to move the idea of building blocks to an industrial scale. Like in other engineering disciplines, software systems shall be assembled from premanufactured components rather than crafted individually by hand. This is an old idea, dating back to the NATO conference on software engineering in 1968 [22]. In 1990 Brad Cox even advocated an industrial revolution in the software realm [10], observing that software components are not just a technological issue but a cultural one as well. In particular, as also stressed in a recent book by Clemens Szyperski [35], the true potential of component software comes from establishing component markets. If system assemblers can acquire individual components from several vendors, they can actually combine the many special skills, ideas, and inventions each vendor has to offer. The number of possibly interesting combinations grows rapidly as vendors can join an open market to offer components in the field of their special competence.

An example, frequently used to illustrate this, are spell-checking components that can be composed with a text editor to provide for in-place checking and correction. For instance, a vendor with special competence in linguistics and a specific language, Finnish, offers an add-on spell checker [18] to be used with Microsoft Word. Offering a component rather than a complete word processor allows the company to concentrate on what their staff is good at. Also, the market for Finnish language checkers is probably not extremely large and may not give enough revenue to finance the construction of a competitive editor.

Furthermore, Microsoft's editor serves as an integration platform with other add-ons offered by further vendors. For instance, a user requiring not only well performed checking of Finnish but also needing to include some special type of diagrams into documents, may not find an of-the-shelf program with this particular feature combination. Custom programming of such a system would be too expensive in most cases, but a custom assembly from standard components may be achievable at a reasonable price. Because of this, component software is also described as a path between standard and custom software [29].

Using industrial components as described above has two requirements. Firstly, using a premanufactured component must be easy enough, compared to programming it from scratch, to balance the cost of acquiring it. Otherwise, component reuse is simply not going to happen because system development costs would increase instead of decrease.

The most inexpensive way to compose components is by plug-and-play. This means that neither the components need to be adapted nor any programming is required to glue the components together. The problem may not even be the programming itself, but the required reengineering and detailed understanding of the components or at least their interfaces. Plug-and-play in turn may sometimes be left to the end user. Netscape's Communicator Plug-Ins [7] are an example of this.

As a second requirement, vendors must be able to produce compatible components despite being mutually unaware. Considering a large component market, it is impossible to demand any vendor to know about all other products and to adapt to them.

It seems, as if the latter requirement of independent component development would contradict the former requirement of plug-and-play, but fortunately this is not the case. Component vendors cannot be expected to synchronize their work with each other, but they can build on common standards. If the latter are properly designed,

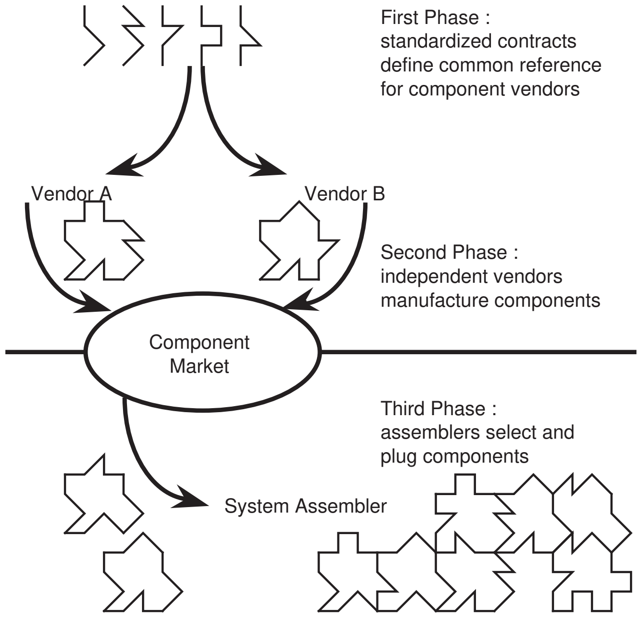


Figure 3: The three phases of component-oriented programming

the independently produced components will still interoperate in a plug-and-play manner.

It is not sufficient, however, to use a common wiring (or plumbing) standard, such as COM, CORBA, or a specific programming language, such as Java. These standards only define the calling conventions for procedures or methods respectively. In addition, component plug-and-play requires application domain dependent standard contracts, specified both syntactically and semantically.

The three phases of creating component systems are shown in Fig. 3, omitting feedback loops, which drive the evolution of standards and components. During the first phase, different standard interfaces are designed and described in public. In the second phase, vendors program towards these standards and place the resulting components on the market. In the third phase, finally, system assemblers select and acquire components from the market and plug them together. Note, that assemblers do not need to analyze *what* a standard interface actually specifies. It suffices to know *that* two components refer to a common standard.

In Java, types are used to support standard contracts. By types we understand both classes and interfaces. Like a plug-and-play system assembler, the loader can check that two components refer to the same type(s) and are thus compatible. A type's name designates the standard. The full behavioral specification associated with it must be stated outside the language in the documentation for component manufacturers.

2.3 A scenario exemplifying a problem with Java

The following example describes a situation that cannot be properly handled by Java's type system. We assume two different and independent standards, which have come into existence entirely unrelated. One of them defines an interface Text, describing operations, such as insertion and deletion of characters. We also assume a transformation function which converts text positions to pixel positions. The second standard defines a compound document framework, like OLE [6], including an interface Container to be implemented by all objects that may act as compound document containers. The latter must support insertion and removal of document parts. Figure 4 shows portions of these two interfaces in Java.

```
/* as part of a text framework: */
  public interface Text {
    void insert (char ch, int textPos);
    /* insert character ch at position textPos */
    ...
    java.awt.Point displayPoint (int textPos);
    /* returns the display position at which the character at textPos is drawn */
    ...
  };

/* as part of a compound document framework: */
  public interface Container {
    void insertPart (DocPart part, java.awt.Point xyPos);
    ...
  };
```

Figure 4: The standard interfaces Text and Container

Both standards form individually useful frameworks. Vendors can build components for either of them. The problem comes with the wish to create components that build on both standards simultaneously. In our example, this would be components that deal with both texts and containers.

In object-oriented programming, combining independently emerged frameworks has been described as an open problem [21]. This is not a problem with component software, where components just implement standard interfaces but do not reuse code from the framework. A component can implement several interfaces belonging to different standard collections. (A common plumbing standard, for instance Java, is still helpful but not strictly necessary.)

```
/* Vendor A's component: */
  public class TextContainerA implements Text, Container {...};

/* Vendor B's component: */
  public class ContainerTextB implements Text, Container {...};
```

Figure 5: Classes offered by vendors A and B

Figure 5 shows portions of two sample classes, TextContainerA of Vendor A and ContainerTextB of vendor B, both implementing the interfaces Text and Container of our sample standards. These classes exhibit a little nuisance. To insert a document part one has to pass the graphical coordinates because the container interface must be used. One may prefer to give a text position and have the part inserted after the corresponding character. For this purpose, a generic service can be implemented that maps the text position to a display position and then inserts the document part there. We assume that a Vendor C wants to offer this service within a class LibraryServices. Figure 6 shows part of this class and Fig. 7 illustrates that whole scenario.

```
  public class LibraryServices {
    public static void insertDocPart (DocPart part, ? into, int textPos) {
      /* the question mark stands for a type saying that interfaces
      Text and Container must be implemented */
      into.insertPart(part, into.displayPoint(textPos));
    }
  };
```

Figure 6: Vendor C's library services

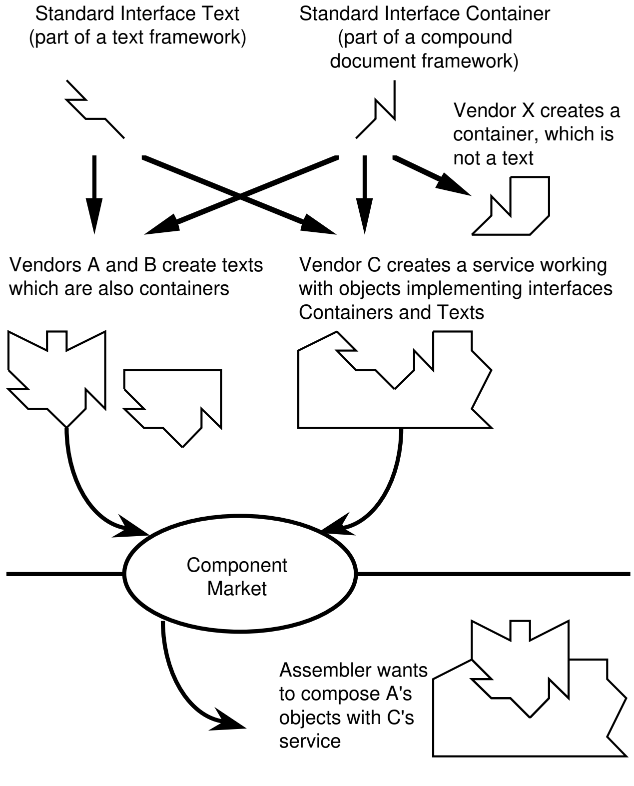


Figure 7: Independent development of classes and insertion service

Vendor C's library service works only for instances of classes that implement both interfaces Text and Container. Unfortunately, this cannot be expressed by the type of parameter into, as the question mark in Fig. 6 indicates.

The obvious solution is to create a combined interface TextContainer, which extends both Text and Container and does not add or hide anything, and to declare parameter into of this type. However, instances of neither TextContainerA nor ContainerTextB are compatible with the library service, as they are both declared to implement only the base interfaces but not the combined interface TextContainer. The problem is who is to define the interface TextContainer to be used by all parties? It is not part of either of the two frameworks because they are assumed to be independent. If one of the vendors A, B, or C defines TextContainer, the others would be obliged to use this definition. This contradicts mutual unawareness postulated for component software vendors. On the other hand, if all three vendors declare their own combined interfaces, they are not compatible either.

The problem can be partly tackled by conventions. A class never implements more than one interface directly; an interface never extends any of its superinterfaces by more than one base interface. Instead, combined interfaces named as concatenation of the fully qualified names of the two direct superinterfaces in alphabetical order are introduced into a package CombinedInterfaces. In our example, all three vendors would create and use interface com_X_Text_com_Y_Container, assuming that Text is part of standard X and Container of Y. The system assembler then deletes all but one of the equivalent definitions. Unfortunately, this renders plug-and-play less feasible.

Furthermore, the conventions-based approach suffers from the combinatorial explosion of the number of interfaces. For the combination of three interfaces, the three pair interfaces have to be created and the combined interface has to be defined as the extension of all three pair interfaces as to make its implementations compatible with the pair interfaces as well. The overhead of this solution and, herewith, the pollution of the name space grows exponentially with the number of combined interfaces. Furthermore, legacy classes that do not abide by this convention are left out.

Finally, this approach fails completely, if we try to combine three or more types one of which is a class. Assume that we have a class C and interfaces I and J. According to the above convention, this would give us the abstract classes CI and CJ as well as the combined interface IJ. For classes implementing all three types C, I, and J to be compatible with the three pair types CI, CJ, and IJ, the triple type would have to extend all three pair types. This, however, is impossible because Java does not support multiple class inheritance. We can define a class D which extends CJ and implements IJ, but instances of D —or D's subclasses— cannot be assigned to variables of type CI (Fig. 8). Java does not permit us to declare a class to be compatible with all subsets of its supertypes. This is an additional problem, not bound to component software.

As a different approach, we can resort to run-time tests and textual annotations. We declare parameter into of InsertDocPart (Fig. 6) to be of type Text, add a comment that it must also implement Container, and cast the parameter's value to Container when accessing the latter's members. In this approach, we loose static type checking.

Yet another possibility would be to use two parameters, one of type Text and one of type Container and require them to reference the same object. Again, we need a less desirable run-time test instead of compile-time type checking.

3 Structure vs. Name Equivalence of Types

The problem described above can be attributed to Java's use of name equivalence of types. Types are compatible only if explicitly declared so. A radical cure would be to use structure equivalence instead, as for instance proposed in [16]. All types that look alike would be considered compatible in this case. From the modeling perspective of object-oriented programming, however, name equivalence is more expressive. In this section we review the advantages of both structural and name equivalence, before we introduce compound types as a beneficial combination of these two in the next section.

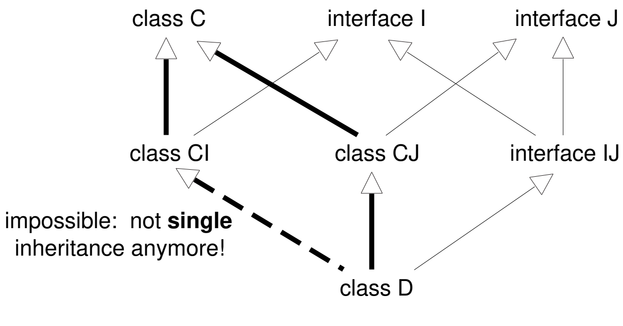


Figure 8: Impossibility of compatibility with all subsets of supertypes

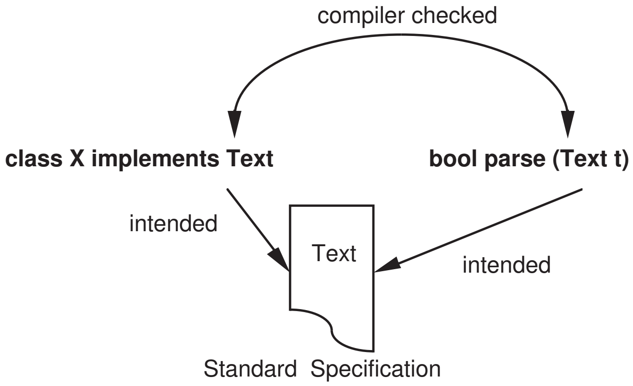


Figure 9: Compiler checked reference to same standard specification

3.1 Structure equivalence of types

With structure equivalence, any two types containing methods and fields with the same names and signatures are equivalent. Likewise, subtyping is based on structure. A type T is assumed to be a subtype of another type S if T contains at least all the methods and fields contained in S. This is the principle; there are more elaborated rules, for instance, allowing for co- or contravariant parameters.

The goal is an as big as possible type matching relation and, therefore, a maximum of flexibility. Types and the relations between them may even be inferred automatically by the compiler and thus need not be declared explicitly by the programmer.

If Java would use structure equivalence between types, the problems described in the previous section would not exist. Vendor C could define a combined interface TextContainer, extending Text and Container and use this to type the parameter into. As both implementations TextContainerA and ContainerTextB contain (at least) all methods named in TextContainer, they would be structural subtypes of TextContainer and, thus, compatible with the library service.

The fundamental purpose of a type system is to prevent the occurrence of run-time errors [4]. On a quasi syntactical level structural type equivalence suffices. Types prevent, for instance, 'method not understood' and memory access violation errors that could occur if, for example, an integer could be assigned to a pointer.

3.2 Name equivalence of types

When using objects as a modeling aid, we would like to eliminate errors beyond those covered by structural type equivalence. Whenever an object of a specific type is required, say as a parameter, we actually intend to require a specifically behaving object. Behavioral subtyping [19] and class refinement [24] formalize this idea of behavior associated to types and of subtypes having to refine the behavior of their supertype(s).

From this point of view, types stand for semantical specifications. While the conformance of an implementation to a behavioral specification cannot be easily checked by current compilers, type conformance is checkable. By simply comparing names, compilers can check that several parties refer to the same standard specification (Fig. 9).

Similarly, Microsoft's COM [32] uses interface identifiers (IIDs) to give each interface its own 'name.' IIDs become, like numbers of ISO standards, abstractions of specifications.

Consider also the analogy to the well-established component market for mass storage with its interface standards such as SCSI, IDE, etc. The buyer of a new hard drive simply ensures that she buys a SCSI drive, if that is what it says on her disk controller. For her, the term SCSI represents a common reference made by the manufacturers of the controller and the drive.

The buyer would not be served well, if she would simply shop for a hard disk with matching mechanical connectors, as a drive that adheres to another, incompatible logical signaling standard might also fit this criterion.

In the same way, even if the projections of two unrelated semantical specifications by coincidence result in the same structure, the two should not be considered equal. As an extreme example of such accidental matches, borrowed from [20], consider two classes: Rectangle with operations Move and Draw and a class Cowboy with operations Move, Draw, and Shoot. Looking only at the structure, Cowboy is a subtype of Rectangle.

This can be ruled out only by forcing programmers to be explicit about their intentions. In other words, type equivalence and subtype relations must be declared rather than be inferred.

To this purpose, several languages use name equivalence. They consider two types compatible only if the declaration of either type explicitly refers to the name of the other. Java is one example. Modula-3 [5] uses structural type equivalence by default, but allows the programmer to explicitly demand name equivalence by assigning a unique brand to a type.

4 Compound Types

Both structural and name equivalence offer benefits as discussed above. Structure equivalence gives more flexibility when composing software, name equivalence allows programmers to better express their intentions. To combine these advantages, we introduce a light-weight construction to explore the middle ground between exclusive use of structure or name equivalence: compound types.

To begin with, let us analyze the respective advantages of name and structure equivalence in the context of our initial example of TextContainers introduced in Sect. 2.3. Name equivalence allows us to explicitly state that objects compatible with interface Text are supposed to adhere to the respective specification, in our example partially provided in the form of comments. A similar statement holds with respect to interface Container. Structure equivalence, on the other hand, would allow us to type the service defined in Fig. 6 more reasonably.

The behavioral specification of that service refers to the two specifications associated with the types Text and Container, not just to the union of the methods and fields defined by these types. In the service's implementation, this shows whenever the parameter into is used as if being of the (behavioral) type Text or Container.

We call a type that combines the behavioral specifications of several other types the *compound type* of these. In the following, we denote a compound type as a list of its constituent types in square brackets. In our example, the service's parameter into would be typed as [Text, Container].

Neither in a language based only on structure equivalence nor in one using only name equivalence such a type can be expressed. With structure equivalence more types than wanted would be compatible because of possible accidental, purely syntactical matches. With name equivalence, different types declared with the same constituent types remain incompatible. Fig. 10 visualizes the different sets defined when using name equivalence, structure equivalence, and compound types.

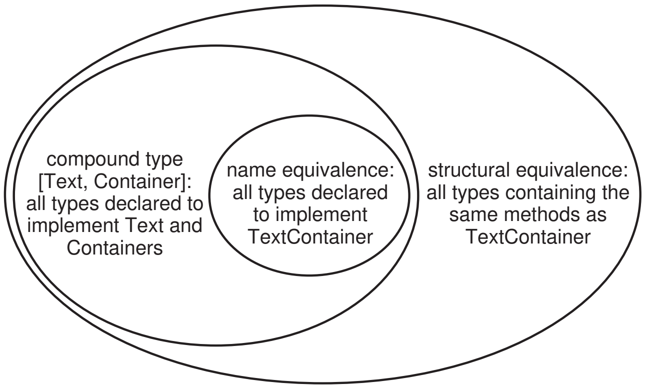


Figure 10: Type compatibility with name equivalence, structure equivalence, and compound types

Compound types, composed from the same behavioral types can be treated as equal even with respect to behavioral specification. Any type subtyping both Text and Container, such as TextContainerA, must respect both semantical specifications at the same time. Consequently, it can be safely cast to either of its constituent types and therefore it is compatible with the corresponding compound type [Text, Container].

We conclude that type equivalence of compound types can and should be defined based on the structure of the composition. We thus speak of structure equivalence of compositions of name equivalent types. Compound types combine the best of two worlds.

Using compound types, we can solve our typing problem of the parameter into from Fig. 6. We give it the type [Text, Container]. Since both classes TextContainerA and ContainerTextB implement the two constituent interfaces Text and Container, instances of them can be passed as actual parameters to the library service. Variable into having all members of its constituent interfaces, the required method calls can be made without any additional casts or run-time validity tests. Thanks to the combination of structural and name equivalence, instances of other classes that just happen to declare methods with the same names and signatures as Text and Container, rather than implement the two interfaces, are rightfully rejected at compile time as values for parameter into. Figure 11 illustrates the subtype relationships, omitting transitive arrows.

Compound types also solve the other typing problem pointed out in Sect. 2.3 and illustrated in Fig. 8. Java's type system does not allow a programmer to declare a class that is assignment compatible with all subsets of extended, respectively implemented types, if the class implements more than one interface and does not have

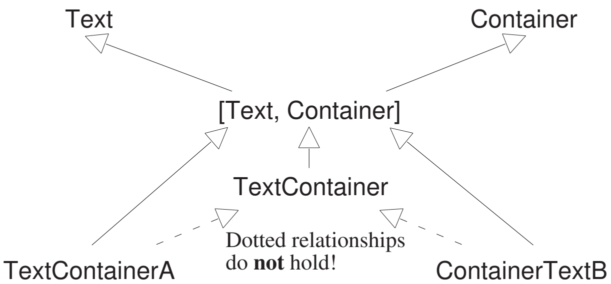


Figure 11: Subtype relationship (transitive arrows omitted)

Object as its direct superclass. Consider now the case with compound types. Let class G extend class C and implement interfaces I and J. Instances of G can be assigned to variables of types C, I, J, [C, I], [C, J], [I, J], and [C, I, J].

Compound types have underpinning in the theory of intersection types ([9, 33], see [30] for a recent overview). The intersection of two types S and T is the type of all elements belonging to both S and T. The new idea is to use defined types, which represent behavioral specifications, rather than the members of these types as atoms.

To mark this specific choice of atoms and to emphasize the intuition of combining specifications, rather than that of intersecting sets of possible values, we have decided to use the new name compound type.

5 Compound Types in Java

In this section we discuss a number of details showing how compound types are integrated into Java. We investigate the conditions for well-formedness and a number of interesting properties.

We define compound types in Java as anonymous reference types. A compound type is a direct extension of a set of interfaces and a non-final class, collectively referred to as constituent types. The members (methods, fields) of a compound type are the members of its constituent types with their respective accessibility. The compound type does not add any additional members, redefine any members, or hide any constants. If no constituent class type is explicitly given, Object is implicitly assumed acknowledging that any reference type can be converted to Object by assignment conversion.

Compound types can be used as parameter types, variable types, return types of methods, cast operators, and operands of the instanceof operator. They are not permitted in the **extends** or **implements** clauses of interface and class declarations.

A variable, the declared type of which is a compound type, may have as its value a reference to an instance of a class declared to extend the constituent class and implement the constituent interfaces, or the value of the variable may be null. In other words, the legal values of a variable, the declared type of which is a compound types, are those that could also be assigned to variables of all constituent types of the variable's type.

Compound types are written as comma separated lists delimited by square brackets. The order of constituent types is not relevant, e.g., [Text, Container] and [Container, Text] denote the same type.

As a guiding principle, a compound type [C, I1, I2, …, In] is well-formed, if the abstract class definition **abstract class** D **extends** C **implements** I1, I2, …, In {}, where D is a fresh name, would be acceptable in the same package. Thus, no two constituent types may define a method with the same name and signature but different return types.

If the statistically rare and, therefore, comparatively minor problem of method clashes were solved in the Java base language, e.g., by qualified names, it would also automatically disappear for compound types.

Including more than one class type is pointless, as no compatible objects could ever be created in Java's single class inheritance system, unless the classes are in a subclass relationship. For simplicity and consistency, we do not allow more than one class to be included.

On the other hand, coherence with Java's design principles dictates that both an interface and one of its superinterfaces may be included. Assume that TrivText is a superinterface of Text. Then, instances of TextContainerA and ContainerTextB can also be assigned to variables of type [Text, Container, TrivText] as their classes indirectly also implement TrivText. However, [Text, Container] and [Text, Container, TrivText] do — due to an in our opinion unfortunate feature of Java — not denote the same type. In Java interfaces may shadow constants defined in their superinterfaces. Let TrivText define a constant **int** k = 21 and Text shadow it by defining **boolean** k = **true**. Then a.k denotes the boolean expression **true** for a of type [Text, Container], but is ambiguous for a of type [Text, Container, TrivText]. In the latter case, a qualification such as Text.a.k would be required. Including an interface that is already implemented by the constituent class is analogous. As in a class declaration, the same interface may not be included more than once in a compound type.

We have introduced compound types as anonymous types only. They could, however, also be given names for documentation purposes. Of course, partially structure equivalence as described above would also apply to the named variant. A named compound type would only be visible where all its constituting interfaces are visible.

The changes we propose to Java's language specification [12] can be found in [3, Appendix].

6 Emulating Compound Types on the Virtual Machine

Our proposed extension requires modifications of the Java compiler, but programs with compound types can be executed on an unchanged virtual machine [17]. The latter is significant because many of the security and portability properties of Java are tied to the virtual machine, as remarked by Agesen et al. [1].

There are —at least— two ways of emulating compound types. Both of them are hinted at in Sect. 2.3. One idea is to use one of the constituent types in place of the compound type and to employ explicit casts to access members of the other constituent types. Most of these casts require run-type validity checks. However, if the byte-code has been generated by a correct compiler all these casts will always succeed at run time, as they have already at compile time been proven correct.

These superfluous run-time tests are also needed when using current Java as only one of the constituent types can be asserted statically. Removing unnecessary tests automatically requires a flow analysis of the complete system. Already expensive for closed systems, this is entirely impossible for extensible systems that by definition are never complete.

Thus, the use of compound types on an existing virtual machine without any adaptation does not incur any performance penalty over a solution in current Java. Rather, if the virtual machine would be adapted to support compound types, a performance increase over the state-of-the-art would result.

The other way is to use multiple variables, one for each constituent type. These variables all contain the same value but have different types. The compiler takes care that the variables are changed in lockstep; a run-time check that all refer to the same object is not necessary. This solution also comes with some overhead in space and time compared to an adapted virtual machine, but it is as efficient as a solution in current Java.

The result of the instanceof expression E **instanceof** [C, L1, L2, …, Ln] is true if the value of E is not **null** and the reference could be cast to [C, L1, L2, …, Ln] without raising a ClassCastException. Let o be a fresh variable of type Object. Then

```
  E instanceof [C, L1, L2, ..., Ln]
≡ (o=E) instanceof C && o instanceof L1 && ...
  && o instanceof Ln
```

In spite of the emulation option on the existing virtual machine, compound types cannot be mapped to plain Java without loosing static safety on the language level.

7 Type Soundness

In this section, we report on a mechanically verified formal proof of type soundness of Java with compound types. Type soundness intuitively means that all values produced during any program execution respect their static types. An immediate corollary of type soundness is that method calls always execute a suitable method, that is, there are no 'method not understood' errors at run time. Type soundness is not a trivial property, especially for polymorphic languages [2, 4]. It came to prominence with the discovery of the failure of its application to older versions of Eiffel [8, 23].

Our proof of type soundness for compound types is based on the work of von Oheimb and Nipkow [36], a much extended version of [26], in which they have formalized and proved type soundness of a large subset of Java. They verified the proof mechanically with the theorem prover Isabelle/HOL [27].

To this formalization, we added compound types as reference types, appended the widening and casting relations with compound types, and defined the members of the latter. Finally, we adapted the proofs and ran them through Isabelle/HOL.[3] The definition of compound types adds 131 lines to the existing 1371 lines, approximately 10 %.

Here, we present the extensions to the widening and casting relations, which are interesting in their own rights. A full report of all the mechanical details is beyond the scope of this paper.

The Java language specification introduces identity and irreflexive widening conversions separately. Since in all conversion contexts permitting widening identity conversions are possible as well, the two are merged in the formalization. The expression $\Gamma \vdash S \preceq T$ says that in program environment Γ objects of type S can be transformed to type T by identity or widening conversion. In particular, expressions of type S can be assigned to variables of type T and expressions of type S can be passed for formal parameters of type T. Widening can be understood as a syntactic, declared form of subtyping.[4] Unlike subtyping in most type theoretic frameworks, the Java language specification does not say that widening is transitive. Hence, transitivity is a proved property rather than an axiom.

We use the following naming conventions:

C, D	classes	M, L	sets of interfaces
I, J	interfaces	S, T	arbitrary types
R	reference type	Γ	program, environment

Likewise, $\Gamma \vdash C \prec_{\mathsf{C}} D$ expresses that C is a subclass of D, $\Gamma \vdash C \rightsquigarrow I$ that class C implements interface I, and $\Gamma \vdash I \prec_{\mathsf{i}} J$ that I is a subinterface of J. Furthermore, is_type Γ T expresses that T is a legal type in Γ, RefT R denotes reference type R, and NT stands for the null type. With this, we can express the following two typing judgments, which are also applicable to compound types:

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

Further Class C stands for the class type C, Iface I for the interface type I, and $\mathsf{T}[\cdot]$ for an array type with elements of type T. Compound (C, L) denotes the compound type with class C and interfaces $L_i \in L$. The discriminators is_class Γ C, is_iface Γ I,

[3] At http://www.abo.fi/~mbuechi/publications/CompoundTypes.html the Isabelle theories are available.

[4] For simplicity, the term 'subtyping' is used in the other sections of this paper in place of the formally correct notion of 'widening'.

$\Gamma \vdash S \preceq T$	S widens to ('is subtype of') T in Γ
$\Gamma \vdash C \prec_{\mathsf{C}} D$	C is a subclass of D in Γ
$\Gamma \vdash C \leadsto I$	C implements I in Γ
$\Gamma \vdash I \prec_{\mathsf{i}} J$	I is a subinterface of J in Γ
$\Gamma \vdash S \preceq_? T$	cast from S to T permissible at compile time in Γ
'no_conflict Γ (I,D,M)'	in Γ interface I, class D, and constituent interfaces of M do not define a method with the same name and the same signature but different return types

Figure 12: Summary of notation

and is_compound Γ (C,L) are also used. The latter is true, if the compound type is well formed, that is, all constituent types are accessible, C is not final, and there is no method name p such that two constituent types define a method named p with identical signature but different return types. We assume that is_class Γ Object and is_iface Γ Cloneable holds for all Γ. In Java, Cloneable is the only interface implemented by arrays. With this we can define the remaining widening rules involving compound types:

$$\frac{\Gamma \vdash \text{Class } C \preceq \text{Class } D;\quad \text{is\_compound } \Gamma\ (D,M);\ \forall J \in M. \Gamma \vdash C \leadsto J}{\Gamma \vdash \text{Class } C \preceq \text{Compound } (D,M)}$$

$$\frac{\text{is\_iface } \Gamma\ I;\quad \text{is\_compound } \Gamma\ (\text{Object},M);\ \forall J \in M. \Gamma \vdash I \prec_{\mathsf{i}} J \lor I = J}{\Gamma \vdash \text{Iface } I \preceq \text{Compound } (\text{Object},M)}$$

$$\frac{\text{is\_type } \Gamma\ T}{\Gamma \vdash \mathsf{T}[\cdot] \preceq \text{Compound } (\text{Object},\{\})}$$

$$\frac{\text{is\_type } \Gamma\ T}{\Gamma \vdash \mathsf{T}[\cdot] \preceq \text{Compound } (\text{Object},\{\text{Cloneable}\})}$$

$$\frac{\Gamma \vdash \text{Class } C \preceq \text{Class } D;\ \text{is\_compound } \Gamma\ (C,L)}{\Gamma \vdash \text{Compound } (C,L) \preceq \text{Class } D}$$

$$\frac{\text{is\_compound } \Gamma\ (C,L);\ \Gamma \vdash C \leadsto J \lor (\exists I \in L. \Gamma \vdash I \prec_{\mathsf{i}} J \lor I = J)}{\Gamma \vdash \text{Compound } (C,L) \preceq \text{Iface } J}$$

$$\frac{\Gamma \vdash \text{Class } C \preceq \text{Class } D;\quad \text{is\_compound } \Gamma\ (C,L);\ \text{is\_compound } \Gamma\ (D,M);\quad \forall J \in M. \Gamma \vdash C \leadsto J \lor (\exists I \in L. \Gamma \vdash I \prec_{\mathsf{i}} J \lor I = J)}{\Gamma \vdash \text{Compound } (C,L) \preceq \text{Compound } (D,M)}$$

The casting relation $\Gamma \vdash S \preceq_? T$ states, that a cast from type S to type T is permissible at compile time, that is, the type cast '(T)e', where e is of type S, might succeed at run-time. If it can be proven to always fail, the compiler can already flag an error.

If $\Gamma \vdash S \preceq T$ holds, the cast can be proven to always succeed. Otherwise, a run-time validity test must be performed to check whether $\Gamma \vdash R \preceq T$ holds for the run-time type R of the cast operand. The following general casting conversions are applicable to compound types as well:

$$\frac{\Gamma \vdash S \preceq T}{\Gamma \vdash S \preceq_? T} \qquad \frac{\Gamma \vdash \text{RefT } S \preceq_? \text{RefT } T}{\Gamma \vdash (\text{RefT S})[\cdot] \preceq_? (\text{RefT T})[\cdot]}$$

In the rules below, 'no_conflict Γ (I,D,M)' means that there is no method name p such that both I and D or one of the interfaces in M declare a method named p with the same signature but different return types.[5] We use this abbreviation freely for different combinations of classes, interfaces, and sets of interfaces to indicate the absence of a method clash in place of the actual predicates, which are lengthy and technical.

$$\frac{\Gamma \vdash D \prec_{\mathsf{C}} C;\ \text{is\_compound } \Gamma\ (D,M)}{\Gamma \vdash \text{Class } C \preceq_? \text{Compound } (D,M)}$$

$$\frac{\Gamma \vdash \text{Class } C \preceq \text{Class } D;\ \text{is\_compound } \Gamma\ (D,M);\quad \neg(\text{is\_final } \Gamma\ C);\ \text{'no\_conflict } \Gamma\ (C,M)\text{'}}{\Gamma \vdash \text{Class } C \preceq_? \text{Compound } (D,M)}$$

$$\frac{\text{is\_iface } \Gamma\ I;\ \text{is\_compound } \Gamma\ (D,M);\ \text{'no\_conflict } \Gamma\ (I,D,M)\text{'}}{\Gamma \vdash \text{Iface } I \preceq_? \text{Compound } (D,M)}$$

$$\frac{\Gamma \vdash D \prec_{\mathsf{C}} C;\ \text{is\_compound } \Gamma\ (C,L);\quad \neg(\text{is\_final } \Gamma\ D) \lor (\forall I \in L. \Gamma \vdash D \leadsto I);\ \text{'no\_conflict } \Gamma\ (D,L)\text{'}}{\Gamma \vdash \text{Compound } (C,L) \preceq_? \text{Class } D}$$

$$\frac{\text{is\_compound } \Gamma\ (C,L);\ \text{is\_iface } \Gamma\ J;\ \text{'no\_conflict } \Gamma\ (C,L,J)\text{'}}{\Gamma \vdash \text{Compound } (C,L) \preceq_? \text{Iface } J}$$

$$\frac{\text{is\_type } \Gamma\ T}{\Gamma \vdash \text{Compound } (\text{Object},\{\}) \preceq_? \mathsf{T}[\cdot]}$$

$$\frac{\text{is\_type } \Gamma\ T}{\Gamma \vdash \text{Compound } (\text{Object},\{\text{Cloneable}\}) \preceq_? \mathsf{T}[\cdot]}$$

$$\frac{\Gamma \vdash D \prec_{\mathsf{C}} C;\ \text{is\_compound } \Gamma\ (C,L);\quad \text{is\_compound } \Gamma\ (D,M);\ \text{'no\_conflict } \Gamma\ (C,L,D,M)\text{'}}{\Gamma \vdash \text{Compound } (C,L) \preceq_? \text{Compound } (D,M)}$$

$$\frac{\Gamma \vdash \text{Class } C \preceq \text{Class } D;\ \text{is\_compound } \Gamma\ (C,L);\quad \text{is\_compound } \Gamma\ (D,M);\ \text{'no\_conflict } \Gamma\ (C,L,D,M)\text{'}}{\Gamma \vdash \text{Compound } (C,L) \preceq_? \text{Compound } (D,M)}$$

Whereas the widening rules can be considered as a particular instantiation of subtyping for intersection types, the rules for casts are believed to be new.

The currently by von Oheimb and Nipkow formalized subset of Java, on which we build, still does not capture all features. Of them final classes, modifiers, class variables, static methods, interface fields, and methods of the class Object would be relevant for compound types.

The main advantages of a mechanized over a paper-and-pencil proof are additional confidence and support for extensions. We

[5] In what is believed to be an omission from the specification [28], Java checks at compile time only for clashes between methods contained in interfaces, but not for clashes between methods contained in classes and interfaces. Opting for maximum static detection of errors, casts involving compound types are defined to check for all kinds of clashes.

would like to stress the second aspect. Not only did the formalization result in a soundness proof, but the proof tool also reminded us of what all needed to be defined about compound types before the desired properties could be established. Most proof scripts worked without modifications. The fact that all theorems were reproved mechanically for the extended language definition conveys more confidence than the typical adaptation of a paper-and-pencil proof with 'this-should-still-hold' handwaving.

8 Related Work

Our analysis leading to the observation that component software demands a combination of named, behavioral types and structure equivalence for compositions of those, was inspired by Microsoft's binary standard COM. To our knowledge, however, it has never been presented on the programming language level so far. Without this underpinning, some existing programming languages offer similar or related constructions. In this section we review in brief Microsoft's COM, the languages Objective-C, Sather, and Modula-3, the theory of intersection types, and —as a quite different technology— binary component adaptation.

Microsoft's COM: The principle idea of using structural type equivalence with named types as atomic building blocks, each presenting a behavioral contract, is very much inspired by Microsoft's Component Object Model (COM) [32, 6]. In COM, objects cannot be accessed directly but through interfaces only. These interfaces have globally unique identifiers (GUID) as names. It is the intention that with each interface also goes a behavioral specification, to be documented separately.

An object's type is defined as the set of the interfaces implemented by it. The COMEL language [14], built to formalize COM, consequently uses interface sets, similar to our compound types, to type objects. This compositional definition of an object's behavior is heavily used, for instance, by the ActiveX framework [6], which defines, for example, an ActiveX control container as any object implementing a specific set of interfaces.

Here the parallel ends, however. Clients of a COM object need to use a separate reference variable to each interface through which they want to interact with the object, because each interface may be implemented by a separate node and thus have a different address in memory. This is acceptable as memory layout under the hood and may be hidden by a proper programming language. We expect such a language to build heavily on compound-typed variables.

To determine whether an object is of a given type, queries must be issued for each interface being part of that type. This may seriously impact a system's performance, in particular, if an object is situated remotely. Therefore, distributed COM (DCOM) introduced a service to retrieve sets of interfaces.

Alternatively, categories could be used. Membership of classes in a certain category can describe, beyond other, that a specific set of interfaces is supported. In this sense, categories can be compared to explicitly declared subtypes. Only if a class makes an explicit reference, that is, registers as a category member, the information can be exploited.

Objective-C: Objective-C [25], an object-oriented extension of C, first introduced the dual class and interface hierarchies. Entities can be typed with a combination of a class type and one or more protocol types (Objective-C's name for interfaces), much like our compound types. Objective-C's type system is not sound; for example, the validity of casts is not checked at run time. Introducing and verifying compound types as part of a type-sound language, such as Java, still remained to be done.

Modula-3: Modula-3 [5] is another language which combines name equivalence and structure equivalence of types. This combination, however, is different than what we proposed. In Modula-3 structure equivalence is the default for all types, unless declared as branded, which makes them clearly distinguishable. Also, Modula-3 supports only single subtyping and thus compound types cannot contribute anything.

Sather: Sather [11], an object-oriented programming language featuring multiple subtyping and subclassing in separate hierarchies, allows the programmer to introduce types as supertypes of already existing classes. That way it offers two symmetric possibilities to introduce a subtype relationship: it can be declared with either the sub- or the supertype. Most other languages require a declaration with the subtype.

The compatibility problem described in Sect. 2.3 may be solved partially by that. Even if vendor C creates the library service after vendor A creates his TextContainer component, C can still declare the type of the parameter into (Fig. 6) in such a way that A's implementation becomes a supertype. This requires, of course, that C is aware of A's component.

Sather allows subtype relationships to be introduced in the source code by programmers of either type, but not by third parties, such a system assemblers, who only have access to the binary components. In our above example, the library service is still not compatible with vendor B's component, as C was not aware of B's implementation and did thus not explicitly declare it to be a subtype. Likewise, any components created after the library service, the manufacturers of which were not aware of the combined type introduced by C, are incompatible with the library service (Fig. 13). With the mutual unawareness postulate for a large component market, Sather's supertyping does, therefore, not solve the problem at hand.

Pure structure equivalence in Java: The use of pure structural type equivalence between classes and interfaces in Java to increase compatibility has been suggested by Läufer et al. [16]. In their suggestion, any instance of a class that provides an implementation for each method in an interface can be used where a value of the interface type is expected. Thus, classes declared to implement several interfaces directly, such as TextContainerA, are compatible with interfaces, such as TextContainer, combined of the base interfaces implemented by the class. However, also classes that by coincidence happen to contain methods with matching signatures but that are not meant to adhere to the associated semantics are assignment compatible. As explained in Sect. 3.2, pure structure equivalence ignores the modeling aspect of types resulting in too large a compatibility relation.

Using only structure equivalence to decide compatibility between classes and interfaces, as proposed by Läufer et al., it is not possible to express that a parameter must also subclass a certain class in addition to implementing some interfaces. As a case in point, structural conformance between classes and interfaces does not solve the problem of compatibility with all subsets of supertypes for the case of three or more types including a class other than Object (Fig. 8).

Furthermore, the proposal requires changes to the Java Virtual Machine, possibly introducing some security problems. In addition, the existing Java language is changed, rather than extended as by our compound types.

Intersection types: As pointed out in Sect. 3, intersection types with classes and interfaces as atoms are the theoretical foundation for our approach. Intersection types were introduced into the λ-calculus in the late 70's by Coppo and Dezani-Ciancaglini [9] and independently by Sallé [33]. The original motivation for introducing intersection types was the desire for a type-assignment system in which the typing of terms is invariant under β-expansion and in which every term with a normal form has a meaningful typing.

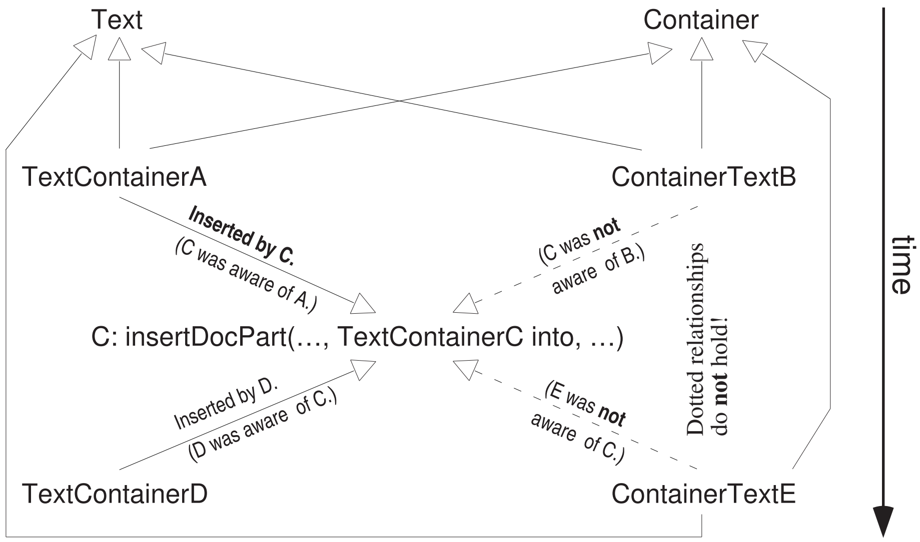


Figure 13: Scenario in Sather where supertyping solves part of problem

In the past twenty years, intersection types, infinite intersections, and the dual notion of union types have been studied extensively in type theory. Pierce and others have also studied the combination of intersection types with bounded polymorphism and other object-oriented concepts (see [30] for a summary of his thesis and an overview of recent work in the field). In contrast to our work, these studies all take the 'type' rather than the 'modeling' view. Thus, they use pure structure equivalence, not taking semantical soundness into account.

Forsythe [31], a descendant of Algol 60, is the only programming language that explicitly uses intersection types and that we are aware of. Forsythe is based on pure structure equivalence, rather than on a combination of name and structure equivalence as our approach. 'Objects' exist in the form of function records only, not allowing for co-variant specialization of the self parameter.

Binary component adaptation (BCA): BCA allows components to be adapted in binary form and during program loading [15]. BCA rewrites class files before or while they are loaded without requiring source code access. Thus, modifications described by delta files can be applied by third-parties. Adding an interface to the implements clause, one of the supported modifications, could be used to solve the compatibility problem described in Sect. 2.3: Vendor C, the creator of the library service, declares a combined interface, which is used to type the parameter into (Fig. 6). Even if vendors A and B have not declared their components to implement this interface, a component integrator can add it to the lists of implemented interfaces using BCA.

BCA adds further flexibility because it can be used to glue classes that are not based on common standard interfaces. Unfortunately, it also burdens the person assembling the system with the task of figuring out how to do this correctly. That is, the system assemblers need to understand the interfaces' semantics and program the adaptation. Plug-and-play with made-to-fit components, as enabled by compound types, is the more economical alternative wherever applicable. Furthermore, BCA makes systems harder to understand as delta files must also be taken into account.

BCA does not solve the problem of compatibility with all subsets of supertypes for the case of three or more types including a class other than Object (Fig. 8), because BCA does not add any new kind of types or modify any conversion rules.

9 Conclusions

We have exhibited a shortcoming of Java's current type system. In a programming language for extensible component software, substitutability of typed objects should neither be decided by the types' name nor just by the structural compatibility of signatures exclusively. Name equivalence, as offered by Java, is too restrictive when composing independently evolved standards or frameworks. Structure equivalence, on the other hand, does not support behavioral typing, that is, to associate semantical specifications with type names.

We concluded that one needs both. On the level of declared types, name equivalence is to be used. A behavioral contract can be associated with each type. When composing these types, however, we want separately declared compositions to be compatible if they have the same structure, that is if they consist of the same types.

To this end, we propose compound types as structurally matched compositions of named types, considered to match only if declared so.

We showed how to add compound types as anonymous compositions of named types to Java, an example of a practical, type-sound programming language. To a variable of a compound type one can assign any object with the same structure in terms of implemented interfaces and extended classes.

Java is well suited to host compound types. Building on multiple inheritance of interfaces, we integrated our proposal smoothly. The resulting language is a strict extension and thus backward compatible. Java programs with compound types can be executed on an unchanged virtual machine.

A mechanical soundness proof gives additional confidence in the well-definedness of the extended type system. The relative ease of adapting a formalization of the existing Java language further illustrates the orthogonality of our proposal. The changes we propose to Java's language specification [12] can be found in [3, Appendix].

We believe that compound types can contribute to any typed language with multiple subtyping and name equivalence of types.

Acknowledgments David von Oheimb and Tobias Nipkow provided us with their formalization of Java and helped us with our extensions. We would like to thank Ralph Back, Dominik Gruntz, Cuno Pfister, and Clemens Szyperski for a number of fruitful discussions. The referees' helpful comments are also gratefully acknowledged.

References

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of OOPSLA '97*, pages 49–65. ACM Press, 1997.

[2] Kim B. Bruce, Robert van Gent, and Angela Schuett. PolyTOIL: A type-safe polymorphic object-oriented language. In *Proceedings of ECOOP '95*, pages 27–51. LNCS 952, Springer Verlag, 1995.

[3] Martin Büchi and Wolfgang Weck. Java needs compound types. Technical Report 182, Turku Centre for Computer Science, 1998. http://www.tucs.fi/publications/techreports/TR182.html.

[4] Luca Cardelli. Type systems. In *Handbook of Computer Science and Engineering*, chapter 103. CRC Press, 1997. http://www.luca.demon.co.uk/Papers.html.

[5] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordanand Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Research Report 52, Systems Research Center, Digital Equipment Corporation, Palo Alto, November 1989. http://www.research.digital.com/src/m3defn/html/.

[6] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[7] Netscape Communications. Netscape Plug-Ins, 1998. http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm.

[8] William Cook. A proposal for making Eiffel type-safe. In *Proceedings of ECOOP '89*, pages 57–70. Cambridge University Press, 1989.

[9] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv. Math. Logik*, 19:139–156, 1978.

[10] Brad Cox. Planning the software industrial revolution. *Software Technologies of the 90's special issue of IEEE Software magazine*, November 1990.

[11] B. Gomes, D. Stoutamire, B. Weisssman, and H. Klawitter. Sather 1.1 : Language essentials, 1998. http://www.icsi.berkeley.edu/~sather/Documentation/LanguageDescription/contents.html.

[12] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[13] Object Management Group. The common object request broker: Architecture and specification, 1997. Revision 2.0, formal document 97-02-25, http://www.omg.org.

[14] Rosziati Ibrahim and Clemens Szyperski. The COMEL language. Technical Report FIT-TR-97-06, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 1997. http://www.fit.qut.edu.au/TR/techreports/FIT-TR-97-06.ps.Z.

[15] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of ECOOP '98*. LNCS, Springer Verlag, 1998. http://www.cs.ucsb.edu/oocsb/papers/ecoop98.html.

[16] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. Technical Report CSD-TR-96-077, Department of Computer Science, Purdue University, 1996.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[18] Lingsoft. Orthografix: Finnish proofing tools for Microsoft Word, 1998. http://www.lingsoft.fi/.

[19] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[20] Boris Magnusson. Code reuse considered harmful. *Journal of Object-Oriented Programming*, 4(3):8, November 1991.

[21] Michael Mattsson and Jan Bosch. Framework composition: Problems, causes and solutions. In *Proceedings of TOOLS USA 97*, 1997.

[22] McIllroy. Mass-produced software components. In Peter Naur, Brian Randell, and J. N. Buxton, editors, *Software engineering: concepts and techniques: proceedings of the NATO conferences. The Conference on Software Engineering held in Garmisch, Germany, 7th to 11th October 1968*. Petrocelli/Charter, 1976.

[23] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[24] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, pages 82–101. LNCS 1313, Springer Verlag, 1997.

[25] NeXT Software, Inc. *Object-Oriented Programming and the Objective-C Language*. Addison-Wesley, 1993. http://developer.apple.com/techpubs/rhapsody/ObjectiveC/.

[26] Tobias Nipkow and David von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, 1998.

[27] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994. See also http://www.cl.cam.ac.uk/Research/HVG/isabelle.html.

[28] Roly Perera and Peter Bertelsen. The unofficial Java spec report, 1997. http://www.nodule.demon.co.uk/java/java-1.0spec-bugs.htm.

[29] Cuno Pfister. Component software: A case study using BlackBox components (online tutorial of the BlackBox Component Builder), 1997. http://www.oberon.ch.

[30] Benjamin C. Pierce. Intersection types and bounded polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, April 1997.

[31] John C. Reynolds. Design of the programming language Forsythe. In *Algol-like Languages*, volume 1, pages 173–234. Birkhäuser, 1997. Also available as CMU-CS-96-146, ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-146.ps.gz.

[32] Dale Rogerson. *Inside COM*. Microsoft Press, 1996. See also http://www.microsoft.com/com/.

[33] P. Sallé. Une extension de la théory des types en λ-calcul. In *Proceedings of Automata, Languages and Programming*, pages 398–410. LNCS 61, Springer Verlag, 1978.

[34] Sun Microsystems, Inc. Java Beans, 1997. http://splash.javasoft.com/beans/.

[35] Clemens Szyperski. *Component Software : Beyond Object-oriented Programming*. Addison-Wesley, 1998.

[36] David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*. LNCS, Springer Verlag, 1998, to appear.