Compound Types: Strong Typing for Architecture Composition

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

Abstract

How can strong typing in programming languages support software engineers to express their architectures? How can composition of architectures be expressed? Architectural roles, to be taken over by components, are behavioral types. This is supported best by programming languages using name equivalence of types. On the other hand, composing architectures works well only if structural equivalence is used. As the way out of this dilemma, we propose compound types, which offer name equivalence for types modeling roles and structural equivalence for the composition of roles.

1 Introduction

Architectures turned into reusable software form frameworks. These frameworks define slots, into which components need to be plugged to instantiate a complete system. The components take over architectural roles defined by the framework.

An excellent aid to model architectural roles is behavioral typing, as used by Liskov et al. [11]. Behavioral types include a semantical specification to be met by any object conforming to the type. Static typing with behavioral types serves to exclude wrong assignments of actors to roles already at compilation time.

For practical purposes, early and reliable error detection means shorter time to market and less expensive production. It is well known, that a problem can be fixed faster and cheaper the earlier in the production process it is detected. Detection at compile time is long before detection at run time, which principally is only by coincidence, even though elaborated testing strategies try to produce such coincidences fast. Still, some problems will show at run time and have to be analyzed then. It is the author's experience that in such a situation it is of tremendous help if one can rely on certain properties asserted by the compiler. This cuts down considerably the set of possible problem causes to be checked.

With current compiler technology, conformance of an implementation to a semantical specification cannot be checked automatically. What can be done, however, is to use types as references to separately provided semantical specifications, in a way similar to the identifiers (names or numbers) of industrial standards. To be effective, this requires a programming language to use name equivalence for types.

When it comes to composition of architectures, one wants to implement components that combine roles from different architectures. This can usually be done by means of multiple subtyping. In addition, one may want to type variables and method parameters to state that a specific combination of roles is required of the object to be assigned. In an open world, in which one wants to combine binary components from separate sources, this constitutes a problem with languages building solely on name equivalence. Structural equivalence would solve this problem for the price of ruining the modeling aid, described above.

This paper is about how to eat the cake and have it too. We introduce compound types as an original mechanism to combine name and structural equivalence of types. Individual interface types are compared based on their names and explicit compatibility declarations, such as subtypes. Compound types are formed as compositions of such atomic types and are matched according to the structure of this composition.

We recently showed how to extend Java with compound types and proved formally the soundness of the resulting type system [1, 2]. A similar mechanism, though not as strongly typed as in Java, can be found in Objective-C [15].

2 Architectural Roles and Strong Typing

The fundamental purpose of a type system is to prevent the occurrence of run-time errors [3]. Strong typing can 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.

With architectures creating slots to be filled, or defining roles to be taken over by components, types should assert that the right actors play the roles. Upon connecting a component to an interface defined by the architecture, types should ensure that the component meets the interface's demands.

As an example consider a compound document framework, such as OLE [5], which defines the roles of containers and document parts, the latter to be inserted into the former. As a specific framework for such an architecture, OLE defines those roles by collections of services to be provided by an appropriate actor. Services in turn are specified as interfaces, listing selectors and signatures, plus a separate, informal description of the expected behavior. The role, and thus the corresponding type, is defined by both these documents together.

With object-oriented modeling, this is well-known. Whenever an object of a specific type is required, say as a parameter, it is actually intended to require a specifically behaving object. Behavioral subtyping [11] and class refinement [14] 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. Whereas the conformance of an implementation to a behavioral specification cannot be easily checked

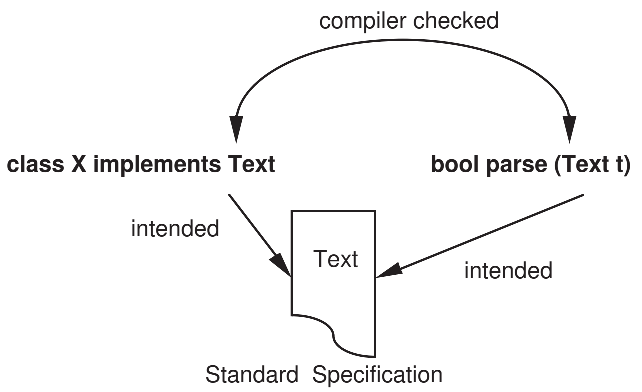

Figure 1: Compiler checked reference to same specification

by current compilers, type conformance is checkable. By simply comparing names, compilers can check that several parties refer to the same specification (Fig. 1).

OLE, for example, builds on Microsoft's COM [18] and, therefore, uses interface identifiers (IIDs) to give each interface its own 'name.' These IIDs are, like numbers of ISO standards, globally unique references to specifications to be found in separate documents.

For further illustration, consider 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 syntactical 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 [12], 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 [4] uses structural type equivalence by default, but allows the programmer to explicitly demand name equivalence by assigning a unique brand to a type.

3 Composing Architectures, Composing Types

In this section, we contrast the observations made in the previous section in favor of name equivalence of types by investigating the needs of framework composition. We focus, however, on the typing issue and do not discuss problems, such as control flow, to be considered when designing architectures and frameworks for composition, as discussed, for instance, in [13].

The following example describes a situation that cannot be properly handled by most implementation languages using name equivalence of types exclusively. We use Java, one such language, to state our example.

We assume two different and independent frameworks having 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 framework is for compound documents, like OLE [5], 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 2 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 2: The standard interfaces Text and Container

Both architectures are useful individually. Different vendors can build components for either one of them. The problem comes with components taking over roles in both architectures simultaneously. In our example, this would be components that deal with both texts and containers.

There are two possibilities of architectural composition: composition of provisions and compositions of demands. Components taking over roles of more than one architecture combine provisions. Services compose demands, if they work only for components combining certain roles.

The first possibility is typically taken care of by multiple subtyping.

Figure 3 shows portions of two sample classes, TextContainerA and ContainerTextB created by two separate, independent, and mutually unaware vendors A and B

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

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

Figure 3: Classes offered by vendors A and B

respectively. Both classes implement the interfaces Text and Container of our sample architectures.

Both 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 third, independent Vendor C wants to offer this service within a class LibraryServices. Figure 4 shows part of this class.

```
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 4: Vendor C's library services

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. 4 indicates.

The obvious solution is to create a combined interface TextContainer, which extends both Text and Container, 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 these were created independently. 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 [20]. On the other hand, if all three vendors declare their own combined interfaces, they are not compatible either.

With current Java, like with any language using name equivalence exclusively, we have to resort to run-time tests and textual annotations. We declare parameter into of InsertDocPart (Fig. 4) 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.

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.

The problem described above can be attributed to the exclusive use of name equivalence of types. Types are compatible only if explicitly declared so. A radical cure would be to use structural equivalence instead, as for instance proposed in [10]. 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.

4 Compound Types

Both structural and name equivalence offer benefits as discussed above. Structural equivalence gives more flexibility when composing frameworks, 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 structural equivalence in the context of our example introduced in Sect. 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. Structural equivalence, on the other hand, would allow us to give a more meaningful type to the service defined in Fig. 4.

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. We shall denote a compound type as a list of its constituent types in square brackets. In our example, the service's parameter into in Fig. 4 would be typed as [Text, Container].

Neither in a language based only on structural equivalence nor in one using only name equivalence such a type can be expressed. With structural 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. 5 visualizes the different sets defined when using name equivalence, structural 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 semantic specifications at the same time. Consequently, it can be safely cast to either of its constituent types and, therefore, rightfully be used wherever the combination of both types is demanded. As compound

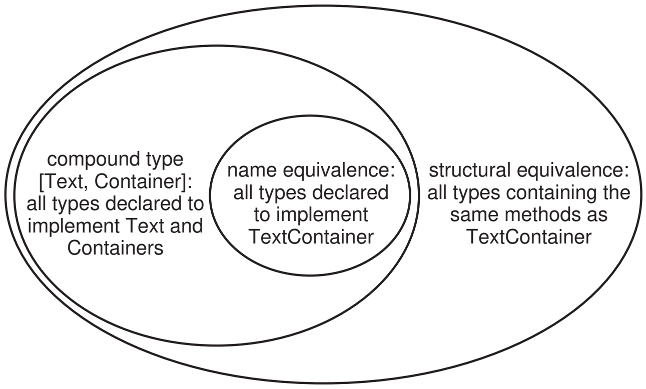

Figure 5: Type compatibility with name equivalence, structural equivalence, and compound types

types do not permit to add, shade, or modify individual methods or fields, each compound type constituted of the same types denotes the same behavior.

We conclude that type equivalence of compound types can and should be defined based on the structure of the composition. We thus speak of structural 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. 4. 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. Figure 6 illustrates the subtype relationships.

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.

Compound types have underpinning in the theory of intersection types ([6, 19], see [16] 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.

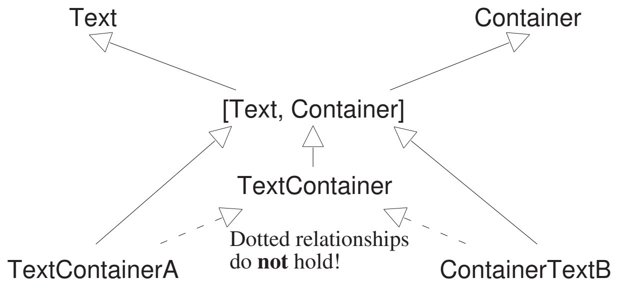

Figure 6: Subtype relationship (transitive arrows omitted)

5 Related Work

Our analysis leading to the observation that composing architectural frameworks demands a combination of named, behavioral types and structural 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) [18]. 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. This compositional definition of an object's behavior is heavily used, for instance, by the ActiveX Component Framework [5], which defines, for example, a visible ActiveX control 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 because it presents the memory layout under the hood only and can 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 have a serious impact on a system's performance, in particular, if an object is situated remotely. Therefore, DCOM specifies a service to query for entire sets of interfaces.

Objective-C and Java: Objective-C [15], an object-oriented extension of C, first introduced dual class and interface hierarchies. Since entities can be typed with a combination of a class type and one or more protocol types, Objective-C features compound types already. However, Objective-C is not strongly typed; for example, the validity of casts is not checked at run time.

Into Java, an example of a type-sound language, compound types can be introduced as a light-weight extension, not invalidating existing programs and not requiring the virtual machine to be changed [1, 2].

Modula-3: Modula-3 [4] is another language which combines name equivalence and structural equivalence of types. This combination, however, is different from what we proposed. In Modula-3 structural 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 [7], 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. 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` in such a way that A's implementation becomes a supertype (Fig. 4). 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. 7).

Intersection types: 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 [6] and independently by Sallé [19]. 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.

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 [16] 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 structural equivalence, not taking semantic soundness into account.

Forsythe [17], 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 structural equivalence, rather than on a combination of name and structural equivalence as our approach. 'Objects' exist in the form of function records only, not allowing for co-variant specialization of the self parameter.

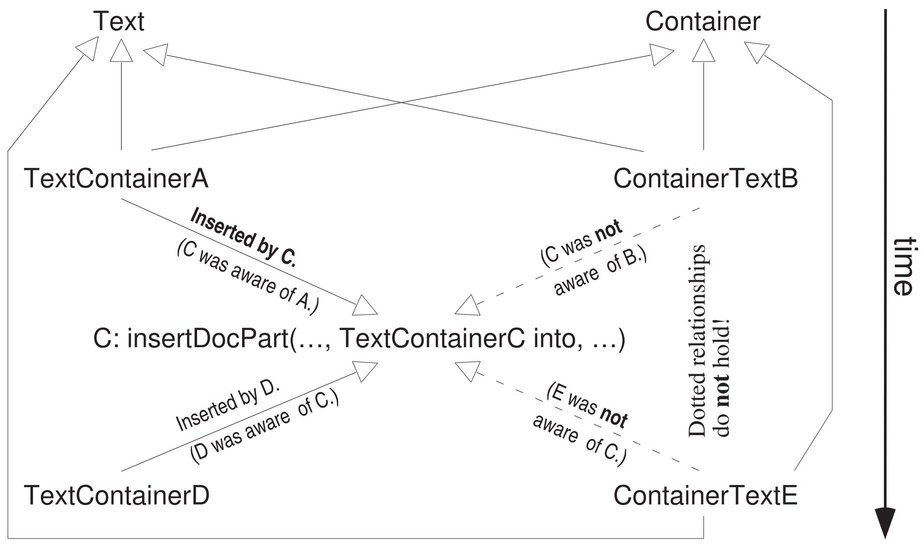

Figure 7: Scenario in Sather where supertyping solves part of problem

Binary component adaptation (BCA): BCA allows components to be adapted in binary form and during program loading [9]. BCA rewrites class files before or while they are loaded without requiring source code access. Adding an interface to the implements clause, one of the supported modifications, could be used to solve the compatibility problem described in Sect. 3: Vendor C, the creator of the library service, declares a combined interface, which is used to type the parameter into (Fig. 4). 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.

6 Conclusions

We argued that a programming language supporting reuse of architectures and frameworks should feature a type system based on name equivalence of types. That way, one can associate externally provided semantical specifications with each type via its name, much like it is done with the names of industrial standards, and a compiler can check that consistent references to such specifications are made.

At the same time, to not hinder composition of frameworks, the language should also use structural equivalence. Otherwise, different, independently working parties would be forced to declare new named types whenever referring to a composition of architectural roles. Without structural equivalence, these types are incompatible and so are the components using them.

We concluded that one needs both name and structural equivalence of types. To model individual architectural roles 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 combine the same roles.

To this end, we proposed compound types as structurally matched compositions of named types.

A report on a extension of Java with compound types, including a soundness proof of the resulting type system, can be found in [1]. The changes we propose to Java's language specification [8] can be found in [2, Appendix].

Acknowledgments We would like to thank Ralph Back, Dominik Gruntz, Cuno Pfister, and Clemens Szyperski for a number of fruitful discussions.

References

[1] Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceeding of OOPSLA'98*, 1998.

[2] Martin Büchi and Wolfgang Weck. Java needs compound types. Technical Report 182, Turku Centre for Computer Science, 1998. http://www.abo.fi/˜mbuechi/publications/CompoundTypes.html.

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

[4] 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/.

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

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

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

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

[19] 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.

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