

## Types and Co-Types in Timor

**J. Leslie Keedy**, Monash University, Melbourne, Australia, University of Newcastle, NSW, Australia and University of Bremen, Germany

**Gisela Menger**, University of Bremen, Germany

**Christian Heinlein**, Aalen University of Applied Sciences, Germany

### Abstract

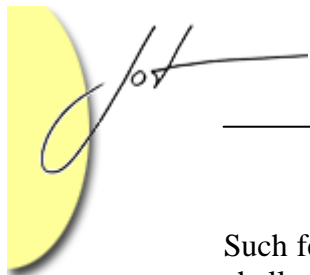
A co-type is a type with instance methods and instance data which enhance the functionality of some other type (its "expanded" type). The instance methods of the co-type correspond approximately to constructors, class methods and binary methods of the expanded type in other class-based OO systems, and the instance data replaces class-based data. This unconventional approach was motivated by the need to reconcile OO concepts with the Timor aims of supporting both multiple implementations for a type and the use of qualifying types.

## 1 INTRODUCTION

About forty years ago Doug McIlroy [20] wrote a widely quoted paper in which he advocated that software should be built in terms of small units (e.g. a sine routine) which can have many different implementations. He envisaged that in this way software houses could compete in providing (small) components and that these components could be built into many different systems. This vision (which in our view differs substantially from that of those who see components as larger structures) is shared by the designers of Timor, and we see the way forward as being realised in an object oriented style, with the help of the information hiding principle, proposed at about the same time by David Parnas [3, 21].

In pursuit of these aims, and in an attempt to introduce greater orthogonality by keeping different concerns separate, Timor has introduced a number of features not usually found in object oriented languages. These include for example

- replacing the class construct with a *type definition* which can potentially have a number of different *implementations* (which may be used in the same and/or different systems and programs) [16, 14], and
- a new kind of component, known as a *qualifying type* [10, 11, 8, 9], which contains *bracket methods* that allow instance methods of other objects to be "qualified" in a modular way, e.g. to protect or synchronise them.



Such features have fundamentally influenced the design of Timor, because they provide new challenges of a structural nature (e.g. with respect to the definition of types). To solve some of these structural issues we have introduced another new kind of type called a "co-type", which itself also supports the aim of component orientation. The purpose of this paper is to motivate and present the idea of co-types.

In the following discussion we make general statements about "conventional" OO languages, without being specific about particular languages. The reader should be aware that by this we mean class-based languages. A discussion of special cases in which attempts have been made to separate types from implementations in class based languages (e.g. via interfaces in Java) has been left to the section on related work.

Section 2 briefly reminds the reader of a few unusual features of Timor which have been presented in earlier papers and which are relevant to the following discussion. Sections 3, 4 and 5 explain why hidden class objects and binary methods are problematic in a language which supports multiple implementations of a type and qualifying types, and proposes their elimination from languages with these features, by replacing them with co-types. In section 6 the properties of types and their co-types are summarised in a general way. Section 7 explains some consequences of co-types. Section 8 describes specific decisions about co-types as found in Timor. Section 9 discusses related work and section 10 concludes the paper.

## 2 IMPORTANT LANGUAGE DESIGN FEATURES

In this section the reader is briefly reminded of some language features of Timor already described in earlier papers, which are not only relevant to the understanding of this paper but which, we believe, will play an increasingly significant role in the design of future object oriented programming languages [1].

### Classification of Instance Methods as Readers or Writers

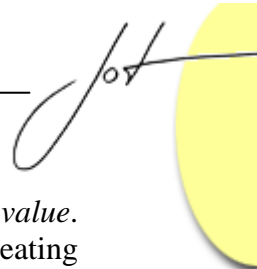
All instance methods are classified in a type definition either as `enq` (enquiries, i.e. reader methods) or as `op` (operations, i.e. writer methods) [14]. This classification provides a basis for achieving such features as reader-writer synchronisation, read-write protection etc. [10, 8].

### Orthogonal Persistence and Orthogonal Distribution

Timor provides the programmer with the appearance of a distributed persistent virtual memory environment [6, 7], which completely hides the existence of conventional file systems and remote computers. We now briefly review how objects are defined and accessed in this memory environment<sup>1</sup>.

---

<sup>1</sup> The implementation is achieved either on a SPEEDOS operating system [4] or by simulating some key features of SPEEDOS.



---

Each implementation of a Timor type has a constructor, which always returns a *value*. This can be assigned to a *value variable* within an object, or it can form the basis for creating a persistent file object or a local object within a file. A file object, which corresponds to either a persistent file or a persistent program in other languages [6], is created by applying the `create` operator to a value. This causes a file object to be created as a copy of the value, and the operator returns the first *capability* for the new file object. This capability can be used to generate further capabilities for the file object. Capabilities can be assigned to variables of the appropriate type and mode. The capability mode is indicated by the `**` notation e.g. `Atype**`. A capability provides its holder with access to the appropriate file regardless of its location in the distributed persistent virtual memory (assuming that it is on-line and that the access rights in the capability permit this).

File objects contain local objects. These can be created from values by applying the `new` operator, which returns the first *reference* for a local object. A reference is a logical pointer, which (unlike a C pointer) cannot be manipulated freely. It is represented syntactically like a C pointer, using the `*` notation. A reference can only be used in the context of the file object in which its corresponding local object appears [6].

A value can be assigned to a variable within a local object. In this case the `*` notation is not used. The values within a local object can be accessed from within that object but are subject to the information hiding principle, and cannot be directly accessed from other local objects within a file object.

A triple asterisk `***` notation indicates that the mode of a variable is a *handle* for a type. This characterises a supertype variable or parameter to which a value, reference or capability of the appropriate type can be assigned. See [6, 7] for a further discussion of persistence and of these concepts.

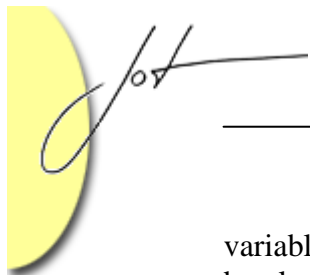
Values, references and capabilities can be declared as abstract variables in type definitions. These are automatically implemented as pairs of setter and getter methods (an `enq` and an `op`) which simply allow the value, reference or capability to be read or written.

If a reference or capability is embedded in an object of the same type (e.g. a reference from an object of type `Person` to another `Person` object) its setter and getter methods are *not* considered to be binary methods in the following discussion.

## Separation of Concerns via Qualifying Types

One important way of achieving separation of concerns in Timor is by the use of qualifying types [10], which have aims similar to those of aspect oriented programming [17, 18, 23], but in a fully integrated manner.

A qualifying type can be instantiated like any other type, and we call objects created from such types *qualifiers*. In addition to having normal instance methods, qualifiers have a number of *bracket methods*. These methods can "catch" a normal method invocation from one object (the *client*) to another (the *target*). The result is that the code of the bracket method is executed in place of (or before) the intended target method, thus allowing the qualifier for example to synchronise or to protect the target. To make this possible a qualifier has its own state variables, which might for example be used to store synchronisation



variables (e.g. semaphores) or protection information (e.g. an access list or a password). The bracket method can use this information to decide whether to invoke the target method (by means of a special keyword `body`) or to refuse access (typically, though not necessarily, by raising an exception).

Different categories of bracket methods can be associated with a qualifier, and these are automatically activated in accordance with the nature of the target method in question. For example the instance methods of a target marked as `enq` or `op` methods can be bracketed by different brackets of the same qualifier, thus enabling reader-writer synchronisation, read-write protection etc. to be achieved. Individual qualification of methods (e.g. on the basis of their method names) is also possible.

The action of a bracket method is invisible to both the client and the target (except in so far as it might raise an exception). Individual target objects can be separately qualified by the same or different qualifiers. Qualifying types are described in more detail in [10] and illustrated in Figure 1:

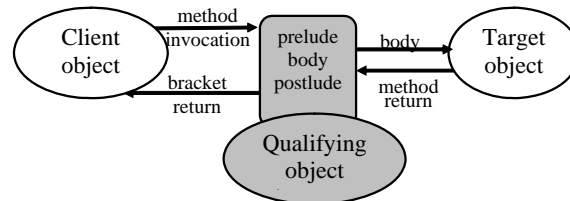


Figure 1: A Bracket Method in Action

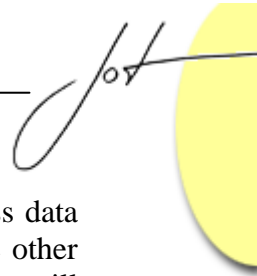
### 3 INITIAL MOTIVATION FOR CO-TYPES

In conventional OO languages such as Java, there is a hidden "class object" for each class used in a program. This object can be used to store "class data" (e.g. "static" variables in Java) and is primarily accessed by constructors (e.g. to count the number of instances created for the class or to hold and increment a serial number for each of the objects in the class) and by class methods (e.g. "static" methods).

In the following discussion we shall see that supporting hidden class objects along the same lines would create some special issues for a language such as Timor, particularly with respect to the possibility of supporting multiple implementations for a type and also the facility to qualify objects.

Binary methods, which we here define as methods that access at least two instances of the class in question, potentially raise some special issues for multiple implementations and for qualifying types, regardless whether they are defined as binary instance methods or binary class methods.

This section explains some of the special issues raised by hidden class objects and by binary methods for a language such as Timor. In some cases these issues could in fact be avoided by simply sharpening the rules compared with languages such as Java and C++, but others require a more radical solution. The solution which we have in fact adopted is to



---

remove all class methods, binary methods, application-oriented constructors<sup>2</sup> and class data from the basic type description and treat them as instance methods and data of some other type. In principle such a type need not have a special role at the language level, but we will show later that there are some advantages of providing a few special constructs to support such types, which we call co-types.

## Issues with Multiple Implementations

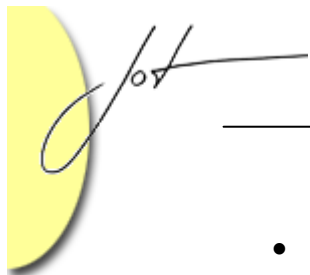
In our view it makes sense to support multiple implementations of the same type, even for apparently very small types, such as a type `Date`, allowing individual dates to be represented in different ways in different systems (or even within a single system). A type `List` provides another example where multiple implementations can be particularly useful. For example lists of varying lengths which hold quite different types of objects and with different usage patterns might usefully co-exist in a single program, implemented in some cases as arrays, in others as linked lists and/or as hash tables, etc. Furthermore one implementation of a type should never need to take into account the existence of other implementations of the same type.

The introduction of multiple implementations within a program raises at least the following questions:

- Should each implementation of the type also implement all the "class" properties? If so, and if for each of these a hidden object is automatically created along the lines of conventional OO languages, the result will for example be multiple counters and serial numbers, one for each implementation of the type.
- However, the application might require a single counter or serial number for use across all instances of the type, regardless of their implementation details. But if only one hidden object is required for the type, in which of the multiple implementations should this be programmed, and how do the other implementations gain access to this?
- Why should it not be possible for the class or type methods themselves to have different implementations, in accordance with the principles of the language?
- How can implementation-oriented parameters be defined for constructors in a type definition? For example, a basic constructor for a list implemented as an array might require a single parameter to indicate how many elements the array should cater for, but a linked list implementation may not require parameters, while for a hash table the basic constructor might require a seed value and a hash table length.
- How can extensive code duplication be avoided in different implementations of the same constructor if the constructor is defined to initialise objects on the basis of application oriented parameters? (Notice that this issue cannot be eliminated simply by defining the initialisation code in a subroutine; this would also have to appear in each implementation.)

---

<sup>2</sup> i.e. constructors which have application oriented parameters that allow each instance to be initialised differently.



- In the implementations of binary methods it would be necessary to take account of the fact that the two or more objects of the type which the binary method manipulates might have different implementations. This would at least mean that the facility provided in languages such as Java for directly accessing the implementation features of both objects would have to be abandoned, and the code of the binary method could only effectively be written by strictly adhering to the information hiding principle, i.e. by accessing each of the objects only via its interface methods.

These issues, while they do not lead inevitably to the conclusion that hidden objects could not be taken over into a language which supports multiple implementations and/or bracket methods, have led us to the decision that hidden objects should not be supported. Instead Timor provides the facility to have explicitly supported objects which serve a function similar to that of class objects.

### Issues with Qualifying Types

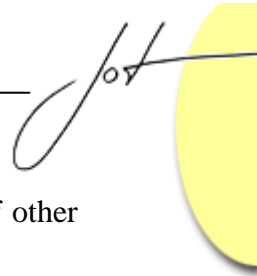
As we saw in section 2, qualifying types operate by "catching" method invocations and substituting bracket methods for these. The latter may then choose to call the intended target by using the `body` keyword. Here are some of the special issues which this raises in connection with the idea of hidden class objects and binary methods:

- Qualifiers can be associated dynamically with target objects, or, in the case of "callout" brackets (see [9]) with client objects. This mechanism relies on the fact that objects can be explicitly constructed, which is not the case with hidden class or type objects. Special conventions would therefore have to be introduced which make hidden objects at least partly explicit.
- The bracketing of conventional constructors is also further hindered by the fact that no object exists at the time a constructor is invoked; it comes into existence as a result of the constructor invocation.
- If binary methods are allowed direct access to the implementation of the objects in question, then this mechanism would allow the effects of qualifiers associated with the individual methods to be by-passed. This could for example lead to incorrect statistics, incorrect synchronisation and to protection breaches. As in the case of multiple implementations a solution to this would at least require that binary methods could not access the internals of their parameters, but would have to invoke their methods in the normal way.

The above points further support our decision to provide co-types as separate types.

## 4 ELIMINATING HIDDEN OBJECTS

A clean solution to the problems associated with hidden objects can be achieved by eliminating such objects entirely from the language. This is possible because almost all the



---

activities which we have discussed can in fact be placed into the code of methods of other types, as the following discussion shows<sup>3</sup>.

## Class/Type Methods

Class/type methods can always be implemented outside the basic type simply by placing them, and the class data, as instance methods (and data) in some other type. The application programmer can determine whether methods are made available to support either implementation-oriented or type-oriented data or both. Furthermore, since these are normal instance methods associated with explicitly constructed objects, qualifiers can then be associated with them in the normal way.

## Constructors

Although constructors cannot simply be placed in another type, our solution in this case is to separate the functionality of a conventional constructor into two parts: a *basic constructor* and a *maker*.

In this organisation each implementation of a type has only one *basic constructor*. Its function is to allocate space for a new object and to initialise its variables to standard values (typically consisting of null and zero values).

Instance methods (which we call *makers*) are defined in some other type as a replacement for each application-oriented constructor. A maker has two basic functions. First it determines what implementation should be used and invokes the appropriate basic constructor. It then uses instance methods of the newly created object to initialise it, as illustrated in Figure 5:

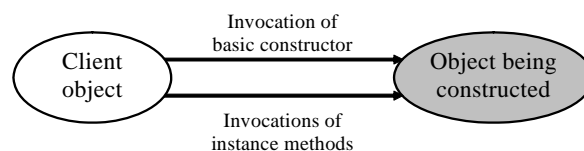


Figure 5: A Maker Method Invoking a Basic Constructor and Instance Methods

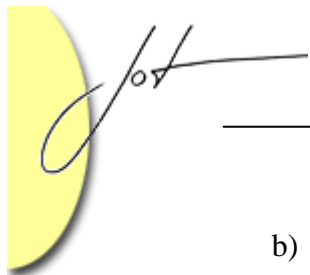
In this organisation the code for the construction phase appears once in each constructor, while the code for the initialisation phase appears once in each maker, thus eliminating the duplications which would occur if constructors were implemented in the standard way, but in each implementation of a type.

This proposal has some further advantages:

- a) The arrangement recalls the organisation of factory patterns [5]. As with factory patterns the initialising method can for example choose which implementation to use and can (but need not) hide the existence of multiple implementations.

---

<sup>3</sup> It is assumed that the basic type has a "complete" set of methods in the sense described in section 6 ("Properties of a Type")



- b) Whereas basic constructors are intended to provide only implementation-oriented parameters, makers need only provide application-oriented initialisation parameters. (Whether implementation-oriented constructor parameters are made visible to the application programmer in direct or indirect form then becomes a decision for the designers of makers.)
- c) Makers can be qualified by bracket methods in the same way as other instance methods. (Conventional constructors cannot be bracketed, because at the time they are invoked no object exists which can be qualified!)
- d) After invoking a physical constructor a maker can organise that the new instance is qualified by bracket methods if appropriate, before the instance becomes accessible to the application.

## 5 ELIMINATING BINARY METHODS

Binary methods, both in the form of binary instance and binary class methods, have long been known to create problems for conventional OO languages (see e.g. [2]). As we have seen above they also raise some new issues in connection with multiple implementations and qualifiers.

In our view the most effective way of eliminating these problems is to eliminate binary methods completely, i.e. to treat them in the way proposed above for the methods associated with hidden objects, i.e. as normal methods of some other type.

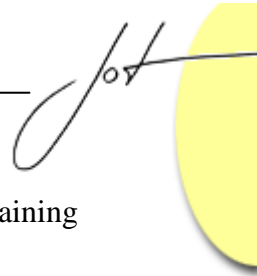
One potential problem might appear to be the case where references to other objects of the same type need to be set or read (e.g. to create a linked list, tree or network of nodes, cf. a genealogical tree of objects of the type `Person`). Although such methods are conventionally classified as binary methods, they have a quite different character from binary methods which actually manipulate or read the contents of two objects of the same type. This difference has enabled us to separate the two cases. In Timor the setter and getter methods for abstract references (see section 2) are treated as an exception to the normal Timor rule that an instance method may not have parameters of its own type. The issue of eliminating binary methods is further discussed in the section on related work.

## 6 SOME PROPERTIES OF TYPES AND THEIR CO-TYPES

We have shown in the previous section that there are good reasons, at least in languages which support qualifying types and/or multiple implementations of a type, for separating the instance methods of a type from other methods, in particular from binary methods, makers (logical constructors) and class methods, which in OO languages are conventionally defined in the same type. This raises the question where should such methods be defined.

In principle they can appear in any other type(s). For example they can be embedded in the code of individual applications, but that is not particularly modular and can quickly lead to similar code being repeated in different application systems. It would also be possible to





---

define types for this purpose in a relatively ad hoc manner, as the idea of factories containing makers [5] shows.

However we believe that it can be helpful to programmers if the language provides a framework for bundling such related methods together into types which are explicitly associated with the types that they expand. This viewpoint has led us to introduce the concept of a *co-type*. In the following discussion we refer to the type which a co-type expands as its *expanded type*.

## Properties of a Type

It follows from the discussion in earlier sections that a type definition consists of a set of instance methods (with a single basic constructor in each implementation), and that these methods must include all the basic functionality needed to allow not only the types but also their co-types to function correctly. For example it is possible in conventional object oriented languages to define constants and initialise them only in constructors. However, with the arrangement suggested above a method must be available to allow makers to initialise each such constant. Similarly sufficient instance methods must be available to allow binary methods to compare instances, etc., cf. the example in section 8 ("Protecting Methods ..."). In this sense one can say that a type consists of a complete set of instance methods needed to manipulate instances of the type, and that an implementation of a type consists of a rudimentary constructor and an implementation of each instance method, together with the appropriate data structures.

## Basic Properties of a Co-Type

Wherever it is reasonable to do so, the number of concepts in a language should be minimised. It therefore seems appropriate to consider a co-type itself as a type in the sense just described above. All the methods which it supports should be bundled together into a single type definition, and an implementation should have a basic constructor together with an implementation of each instance method and the instance data needed to support the implementation.

This approach has a number of advantages. Apart from being able to have multiple implementations, a co-type instance can for example be bracketed by qualifiers; it can inherit and be inherited, and so on. All this comes at no extra cost.

Nevertheless it raises some issues which require further discussion.

## 7 SOME CONSEQUENCES OF CO-TYPES

In this section we discuss some interesting issues raised by the concept of co-types.

## Modularity

We envisage that type definitions and their implementations can (and hopefully will) be programmed for use as components which can be incorporated with minimum effort into different application systems (e.g. by a new generation of software houses, which we refer to as "component software houses").

In this context we see an immediate advantage of separating types (and their implementations) from co-types (and their implementations). It becomes possible to develop (i.e. define and implement) more than one co-type to expand the same type. This implies that a type can be expanded in different ways by several co-types. This point will be discussed in more detail below.

## Factories

We have defined makers as methods designed to invoke a basic constructor and then carry out the application-oriented initialisation phase of conventional constructors, and we have suggested that they can play a role similar to that envisaged for factory methods in creational patterns [5]. Hence the basic functionality of makers can be viewed as providing a convenient interface which

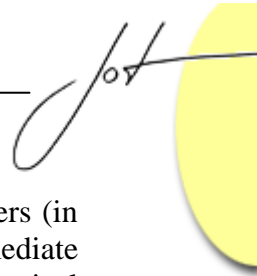
- allows application programs to create new instances of expanded types, and
- can be used (where appropriate) to control such creational activities centrally.

### *Creation of new instances:*

To create a new instance a maker normally invokes an appropriate constructor of an implementation of its expanded type. This implies that decisions regarding the choice of implementation can be hidden in makers. Assuming that several implementations are available in a program and that instances can only be created by makers, this means that implementation choices can be confined to a single module (and therefore that such decisions are relatively easy to change). For example the introduction of new implementations for an expanded type into an existing system needs changes to be made only to the makers for that type.

What criteria should be used to select an implementation when creating a new instance of an expanded type? An important decision for designers/programmers of makers is whether (and to what extent) implementation dependent information is exposed to the application programmer.

At the one extreme all implementation details can be hidden in the maker, so that for example maker implementations select which implementations of the expanded type are to be invoked and determine all the parameter values needed by their constructors. This need not simply be based on fixed values programmed into a maker, but for example decisions might be based on which process is invoking the maker, or on the basis of statistics previously gathered by the co-type, etc. In this case the repeated code problem mentioned in section 3 is completely avoided.



---

At the other extreme all implementation parameters may be made visible by makers (in which case the code repetition problem is not automatically eliminated). Or intermediate solutions are possible, e.g. by providing maker parameters to indicate the expected logical size of an object such as a list, helping the maker to choose between say an array or a linked list implementation, without the application programmer having to make an explicit decision, or even being aware which implementations are available. Hence the distinction between implementation-oriented constructors and separate makers allows a wide range of flexible design decisions to be made.

#### *Central Control:*

So far it has been assumed that co-types have no special privileges. This implies that programs need not use co-types and, if co-types containing makers exist, application programmers can ignore them, e.g. by directly calling the constructor of an implementation. For ad hoc systems and one-off programs this is probably the best solution, so that a programming language should certainly not be designed with a built-in decision that only makers can directly call constructors. (This would also lead to the recursive situation that each co-type must have a co-type!)

However, in professional systems and programs (e.g. involving many programmers) a good case can be made out for controlling this situation. Only in this way can it be guaranteed, for example, that makers can maintain and use correct statistics about the frequency of use of particular implementations when a new instance is created for an expanded type. As another example, only by the compulsory use of makers can it be guaranteed that each instance is, for example, initialised with a unique serial number (e.g. a passport number or student matriculation number) and/or with appropriate bracket methods.

A mechanism which ensures central control can also be used to enforce alternative design decisions, e.g. the creation of new instances of a type by prototyping, the enforcement of singleton instances, etc.

These points suggest that language designers should consider how to provide a mechanism which can (but need not) be used to enforce the use of makers.

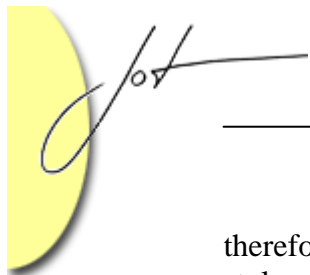
A similar issue arises as a result of the need for an expanded type to provide a "complete" set of instance methods, as described in section 6 ("Properties of a Type"). If an instance method is provided which is intended only for use by a co-type to initialise constants, how can the use of this method be restricted?

These are questions which may have different answers in different languages. We will describe the Timor solutions in section 8.

## Invocation of Co-Type Methods

Conventional OO languages need a special syntax for invoking constructors and class methods (including binary class methods). The standard way of doing this is to use the class name instead of the instance name.

However for co-types no additional syntax is in principle necessary, because the methods are instance methods, and so can be invoked directly via an instance in the normal manner. It



therefore becomes a matter of taste whether language designers imitate the conventional style.

## Co-Type Instances

In a language such as Java, there is one hidden class object per class. Are similar restrictions necessary for co-types? Here we must distinguish between three different issues.

*May a co-type have several implementations?*

Here the answer must be positive if we support the idea of components, especially for use in different systems.

*May a co-type have several instances in a single program or system?*

The answer to this question depends largely on what functionality the co-type contains. For example if its purpose is to provide an alternative for static data and this is on a once per type basis, e.g. to count the number of instances of an expanded type, then only one instance will be necessary. On the other hand a co-type for `Person` if used in a government registry office system could provide makers which create `Person` objects to reflect births, binary methods such as `marry` and `divorce` corresponding to relationship records concerning marriages and methods which record deaths. It could then be appropriate to have different instances of the co-type for different marriage registry offices. So in the general case there appears to be no strong reason to forbid multiple instances of a co-type.

*May several different co-types co-exist for a type?*

In the interests of modularity and extensibility of systems the existence of multiple co-types for a type can scarcely be rejected. On the contrary this allows co-types with different functionality to be programmed separately (e.g. for a type `Person` binary methods handling marriages can be separated from those for carrying out comparisons), and new co-types can be later added to a system without requiring any re-write of existing code.

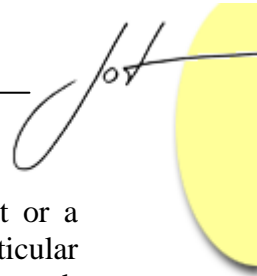
## Enhancing Qualifying Types

As indicated earlier, qualifying types support bracket methods which can modify different categories of methods (e.g. `op` and `enq` methods) in different ways. Despite the fact that they are considered in this paper to be instance methods, it can be quite useful to place makers and binary methods into special categories from the viewpoint of bracketing.

By recognising special categories for makers and for binary methods, code which would otherwise have to be programmed individually in each such method can sometimes usefully be delegated to a qualifying type. As a simple example, statistics about the number of times makers are successfully invoked can be coded once, instead of in several makers<sup>4</sup>. In fact such a qualifying type could be associated with all makers in all co-types, thus indicating how many instances have been created in the entire system. Similar considerations apply to binary methods.

---

<sup>4</sup> Bracket methods can recognise whether or not a maker invocation is successful by checking if the returned value is null.



---

In another scenario a protection qualifier (e.g. which uses an access control list or a revocation list, see [10]) can prevent certain processes from calling the makers of a particular type (or makers of any type) and thus prevent certain users from creating objects. In such cases it would be tedious for a programmer to have to define and implement separate bracket methods for each maker in the target type. In fact component software houses would not need to be aware of the names of methods in a target when developing such a co-type.

These examples suggest that makers and binary methods should be identifiable as separate categories from the viewpoint of bracketing.

### Additional Compile Time Checks

If binary methods and makers are recognised as separate categories of instance methods within a co-type, this is not only useful for developing qualifying types, but can also be used to provide additional compile time checks.

For example, the result returned by a maker should always be defined to have the type which the co-type expands. Similarly a binary method will always have at least two parameters of the expanded type. If co-types are explicitly designated then such details can be checked at compile time.

### Inheritance of Co-Types

Co-types can have subtypes, because they are themselves (almost) normal types. However, because of the special relationship between a co-type and its expanded type a subtype of a co-type must expand the same type as that of its supertype.

Notice that this does not lead to parallel hierarchies between the subtypes of a co-type and the sub-types of the expanded type. For example a co-type `Persons`, which expands a type `Person`, might have a subtype `Persons2`. This is also a co-type for `Person`, but it is *not* a co-type for `Student` (a subtype of `Person`).

### Summary

The above discussion suggests that co-types can provide a useful contribution to an object-oriented approach to component development. Although some features of co-types could (like factories) be achieved without any additions to a language, we have also seen that recognising co-types explicitly in a language can add further benefits which enhance the work of a software developer and/or an application programmer. In the next section we illustrate how co-types have been integrated into Timor.

## 8 DEFINING CO-TYPES IN TIMOR

In accordance with the usual Timor style used in similar cases (e.g. qualifying types) a co-type has the following structure:

```

type C expands T {
  // the keyword expands indicates a co-type relationship
instance: // basic instance methods for the type C,
  // which can (but need not) be used as type methods for T
maker:      // makers for type T
binary:     // binary type methods for type T
}

```

The order of sections is not important, and multiple sections of the same kind can appear in a type definition. Here is an example which we will elaborate in later sections:

```

type Persons expands Person {
instance:
  enq int instances();
  // returns the count of Person instances
maker:
  op Person* init(String name, address; Gender gender;
                  Date dateOfBirth) throws InvalidParams;
  // carries out consistency checks and where appropriate
  // creates a Person value, converts this to a local object,
  // initialises name, address, gender and date of birth,
  // adds an automatically generated identification number
  // and increments the count of Person instances
binary:
  enq boolean equal(Person*** p1, p2) throws NullPtr;
  // compares two Person instances using their unique
  // identification numbers
  enq Person*** older(Person*** p1, p2) throws NullPtr;
  // returns the older of two Person instances
}

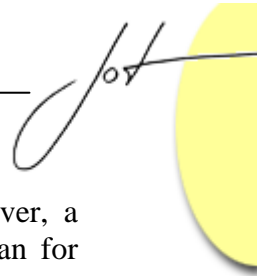
```

## Defining and Implementing Makers

In Timor, if makers are defined for a type, then these are the only methods which can invoke constructors of the implementations of the expanded type. If a type has no makers in any of its co-types the basic constructors for its implementations can be invoked without restriction. Thus the (optional) existence of makers always implies control of a type. We therefore refer to a co-type which contains makers as a *controlling co-type*.

Makers must return an instance of the expanded type. Any of the modes of the type, including handles (see section 2), can be used. If an attempt is made to define a result of a different type, a compile-time error occurs.

If a maker exists for an expanded type which is defined to contain "constants", then only the makers and other methods of the co-type (e.g. co-type instance methods designed to allow controlled changes) can invoke the `op` method for setting the constant. (How this is



---

achieved is described in the subsection below on "Protecting Methods ..."). However, a controlling co-type is not restricted to doing this only once, which means that it can for example make controlled changes to "constants" after the initialisation phase, e.g. to correct errors, or to change a surname after marriage. (In this sense Timor does not support genuine constants, except where the type has no controlling co-type.)

## Defining and Implementing Instance Methods

Instance methods follow the same rules in co-types as in other types. In the context of a co-type they usually access instance data structures which typically serve a function similar to class data in conventional object oriented languages. In this case they will typically not have parameters of the expanded type.

However instance methods can be used to provide other useful co-type functions, in which case they may have a parameter of the expanded type. For example a method which allows a `final` abstract value of the expanded type to be modified for a particular instance (see the previous subsection on makers) needs an actual parameter indicating to which instance the new value applies.

## Defining and Implementing Binary Methods

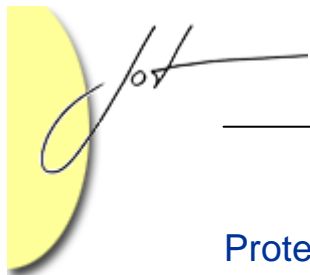
A binary method must have at least two parameters of the expanded type, or a compile time error occurs. It is recommended that these be declared as *handles* for the expanded type, thus ensuring that separate binary methods do not have to be written for accessing instances of the type which have different (or even mixed) modes.

Two examples are shown above, in the definition of the type `Persons`. Although it will generally not be necessary, an implementation of a binary method (like any other method) can if necessary use a cast statement to determine the actual modes of the parameters passed to it. Although in the above example it may appear that only *objects* of type `Person` can exist (because the maker is defined to return an instance of the type `Person*`), instances which are values appear in the code of the maker itself (before they are converted into objects), and so the use of handles ensures that in this maker code the binary methods can be invoked to carry out comparisons (e.g. between a new value and an existing object) before the maker commits to the creation of a further object.

## Singletons

As was indicated in the discussion of co-type instances in section 7 there are examples where several instances of the same co-type can make sense within a single program, while in many cases (especially where the state data provides a functionality similar to that of class data in conventional OO languages) a singleton instance will often be more appropriate.

In any type definition (not just for a co-type) a programmer can designate a type as a singleton by placing the keyword `singleton` at the beginning of the definition. This has the effect that only one instance of this type can be instantiated in each persistent file object (see [6]) – hereafter referred to simply as a "file", including a program file.



## Protecting Methods of an Expanded Type Accessible Only to Co-Types

In section 6 ("Properties of a Type") we indicated that a type definition should include all the methods needed to allow their co-types to function correctly. This is not always the case with definitions of types designed purely for the normal client. For example, a stack type is often defined as simply having methods such as `push`, `pop`, `top`, and `isEmpty`. However a binary method of a co-type which compares two stacks can be much more efficiently implemented if it can access each element in the stack in turn. This can be achieved via a method, e.g.

```
protected enq ELEM getEntryAtPos(int position);
```

The keyword `protected` is used in this context to give a co-type access to protected methods of its expanded type.

## 9 RELATED WORK

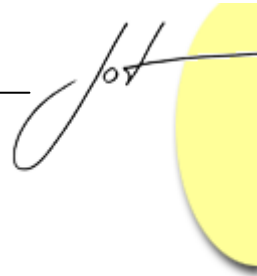
The work most closely related to the present paper is contained in the Ph.D. dissertation of Schmolitzky [22], who was previously a member of our research group in Ulm. This was one of the foundations upon which Timor has been designed. It provides a thorough discussion of the idea that an object oriented programming language should distinguish between a type and potentially multiple implementations. However, it envisaged a construction which differs considerably from a co-type. This is not a different type, but corresponds to an explicit version of a "type object" which cannot have multiple implementations and which is closely bound with the basic type. This has "type operations", which include binary methods and methods corresponding to class methods. Although there is no direct equivalent to makers, "abstract constructors" can be defined in a type definition. These define only application-oriented parameters, and an implementation for an object, together with implementation parameters, can be selected by means of pragmas.

### Removing Constructors/Makers from Type Definitions

In the context of the Theta language Barbara Liskov [19] drew attention to the problem of including constructors with type definitions. The Theta solution was to place them outside type definitions, but not in related types equivalent to Timor co-types. This solution does not distinguish between application- and implementation-oriented parameters.

If we ignore the fact that in Java classes are themselves types and instead view a Java interface as a type definition and a class as an implementation, then one can argue that Java (like Timor) has entirely removed constructors from types. However, interfaces are not compulsory and in many programs the only types which are used are classes, which have constructors that mix logical and implementation parameters. The special Java solutions for avoiding code duplication in constructors (via initialisation blocks and or nested constructor calls) are limited to the implementation of multiple constructors in a single class and therefore do not apply to Timor, which avoids these special mechanisms.





---

## Creational Design Patterns

A key aim of factory design patterns [5] is to place within a single module the decision to use a particular implementation (e.g. deciding which kind of implementation is appropriate in a particular application context, or which of several implementations should be used for particular objects), thus separating application modules from implementation decisions. In Timor applications the realisation of this aim is ideally suited to co-types. The designer of makers for a co-type can choose to hide the implementation related parameters of constructors and can force all parts of an application to use a particular implementation, for example on the basis of the configuration environment in which the application runs. Or he can enforce the use of different implementations for different purposes.

In contrast with other languages, Timor guarantees that application programmers must call the factory methods in order to create new objects, which is especially important in a large system involving many programmers.

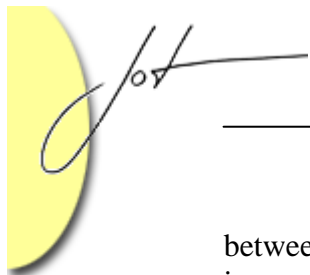
## Avoiding Binary Methods

In their detailed discussion of binary methods (by which they primarily had in mind binary instance methods) Bruce et al. [2] emphasized in particular the difficult issues which arise when attempting to reconcile subtype polymorphism, code re-use and covariant adjustment of parameters. Among the potential solutions which they explored was to avoid binary instance methods completely. This is in fact the solution adopted in Timor, where binary methods are removed from the type definition into a co-type. However, in contrast with the solutions which Bruce et al. explore, Timor co-types maintain a close relationship with their expanded types.

A significant outcome of this approach, as we illustrate in detail in a companion paper [13] is that via the use of co-types the kind of automatic covariant adjustment of parameters which Bruce et al. sought to achieve can be fully realised for both makers and binary methods. At the same time neither subtype polymorphism nor code re-use for co-types and their expanded types needs to be sacrificed.

## 10 CONCLUSION

Initially motivated by problems which potentially arise in a language that supports multiple implementations of a type and qualifying types with bracket methods, the paper presents a new concept, co-types. These allow other types to be expanded by adding makers, binary methods and the equivalent of class methods. These additional methods are in fact instance methods of the co-type and can therefore be treated as such for many purposes. Hence other Timor techniques, such as multiple implementations for a type [14], code re-use [16, 12, 15] and bracketing by qualifying types [10], can be applied to co-types without additional overhead or language complications, except of course in so far as the co-type mechanism itself adds to the tasks which a Timor compiler has to carry out. Recognizing the differences



between makers, binary methods and instance methods at the syntax level also allows improvements to bracket methods and facilitates further compile-time checks.

At the same time co-types provide a convenient structure for implementing factory methods, and they do so in such a form that applications are forced to use a factory method, if one exists.

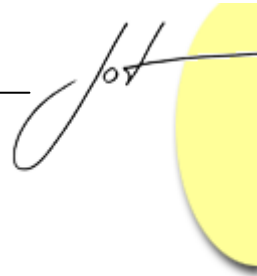
Separating makers from constructors allows implementation parameters in constructors to be separated from logical initialisation parameters and gives maker designers the freedom to hide implementation parameters, and thus avoid potential code repetition problems which might otherwise occasionally arise, freeing application programmers from the need to be aware of the existence of different implementations for a type.

Instance methods of co-types can be used to provide the equivalent of class methods in other languages. State data of co-type instances can conveniently store both what would conventionally be class data and also additional information about instances of the expanded type. The decision to have only one instance of a co-type can be determined by declaring this as a singleton.

By providing for binary type methods in co-types both binary instance methods and binary static (type) methods can be eliminated from types, thus eliminating a number of problems which conventional binary methods can create. In co-types they do not raise these problems because they are simply instance methods of a different type.

Despite the fact that we were initially motivated by a need to solve problems which arise from special features in Timor (multiple implementations of a type, qualification of types) the result appears to be much more generally applicable. Future OO language designs without these special features could also greatly benefit by incorporating co-types, since these provide a structured approach which eliminates some weaknesses of current OO programming languages, for example, in the areas of creational patterns, binary methods and class data.

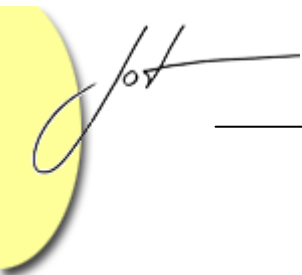
Finally we have indicated in this paper that a subtype for a co-type becomes an additional co-type for the expanded type of its ancestor. However we left open the issue whether some other relationship might exist between co-types and subtype hierarchies for their expanded types. In a companion paper [13] we discuss this possibility and show that a technique similar to, but not identical with, inheritance and subtyping can usefully be applied to co-types to produce co-types for the subtypes of expanded types. This technique, which we call "adjustment", can covariantly adjust the parameters of co-types safely, for example to produce a co-type for a type `Student`, based on a co-type `Persons` which expands `Person`, a supertype of `Student`.



---

## REFERENCES

- [1] J. Brauer, C. Crasemann and H. Krasemann, *Auf dem Weg zu idealen Werkzeugen - Bestandsaufnahme und Ausblick*, Informatik Spektrum (2008), pp. 580-590.
- [2] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens and B. Pierce, *On Binary Methods*, Theory and Practice of Object Systems, 1 (1995), pp. 221-242.
- [3] P. J. Courtois, F. Heymans and D. L. Parnas, *Concurrent Control with Readers and Writers*, Communications of the ACM, 14 (1971), pp. 667-668.
- [4] K. Espenlaub, *Design of the SPEEDOS Operating System Kernel*, Ph.D.Thesis, Dept. of Computer Structures, University of Ulm, 2005, pp. [http://vts.uni-ulm.de/query/longview.meta.asp?document\\_id=5333](http://vts.uni-ulm.de/query/longview.meta.asp?document_id=5333).
- [5] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [6] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, *Persistent Objects and Capabilities in Timor*, Journal of Object Technology, 6 (2007), pp. 103-123 [http://www.jot.fm/issues/issue\\_2007\\_05/article3](http://www.jot.fm/issues/issue_2007_05/article3).
- [7] J. L. Keedy, K. Espenlaub, C. Heinlein and G. Menger, *Persistent Processes and Distribution in Timor*, Journal of Object Technology, 6 (2007), pp. 91-109, [http://www.jot.fm/issues/issue\\_2007\\_07/article2](http://www.jot.fm/issues/issue_2007_07/article2).
- [8] J. L. Keedy, K. Espenlaub, C. Heinlein, G. Menger, F. Henskens and M. Hannaford, *Support for Object Oriented Transactions in Timor*, Journal of Object Technology, 5 (2006), pp. 103-124 [http://www.jot.fm/issues/issue\\_2006\\_03/article1](http://www.jot.fm/issues/issue_2006_03/article1).
- [9] J. L. Keedy, K. Espenlaub, G. Menger and C. Heinlein, *Call-out Bracket Methods in Timor*, Journal of Object Technology, 5 (2006), pp. 51-67, [http://www.jot.fm/issues/issue\\_2006\\_01/article1](http://www.jot.fm/issues/issue_2006_01/article1).
- [10] J. L. Keedy, K. Espenlaub, G. Menger and C. Heinlein, *Qualifying Types with Bracket Methods in Timor*, Journal of Object Technology, 3 (2004), pp. 101-121, [http://www.jot.fm/issues/issue\\_2004\\_01/article1](http://www.jot.fm/issues/issue_2004_01/article1).
- [11] J. L. Keedy, K. Espenlaub, G. Menger, C. Heinlein and M. Evered, *Statically Qualified Types in Timor*, Journal of Object Technology, 4 (2005), pp. 115-137, [http://www.jot.fm/issues/issue\\_2005\\_09/article5](http://www.jot.fm/issues/issue_2005_09/article5).
- [12] J. L. Keedy, C. Heinlein and G. Menger, *Reuse Variables: Reusing Code and State in Timor*, 8th International Conference on Software Reuse, Springer Verlag, Berlin, Madrid, 2004, pp. 205-214,



---

<http://www.springerlink.com/content/vh3515ulhhmyk39x/?p=bac45e4a92a2433f9a6bb13aad40781b&pi=12>.

- [13] J. L. Keedy, G. Menger and C. Heinlein, *Covariantly Adjusting Co-Types in Timor*, (2009), pp. (to appear in th next edition of JOT Vol 9. No. 1, January-February 2010).
- [14] J. L. Keedy, G. Menger and C. Heinlein, *Inheriting from a Common Abstract Ancestor in Timor*, *Journal of Object Technology*, 1 (2002), pp. 81-106, [www.jot.fm/issues/issue\\_2002\\_05/article2](http://www.jot.fm/issues/issue_2002_05/article2).
- [15] J. L. Keedy, G. Menger and C. Heinlein, *Inheriting Multiple and Repeated Parts in Timor*, *Journal of Object Technology*, 3 (2004), pp. 99-120, [http://www.jot.fm/issues/issue\\_2004\\_11/article1](http://www.jot.fm/issues/issue_2004_11/article1).
- [16] J. L. Keedy, G. Menger and C. Heinlein, *Support for Subtyping and Code Re-use in Timor*, in J. Noble and J. Potter, eds., *40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, *Conferences in Research and Practice in Information Technology*, Sydney, Australia, 2002, pp. 35-43.
- [17] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm and W. G. Griswold, *An Overview of AspectJ, ECOOP 2001 - Object-Oriented Programming*, Springer Verlag, LNCS, 2001, pp. 327-353.
- [18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-Oriented Programming, ECOOP '97*, 1997, pp. 220-242.
- [19] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson and A. C. Myers, *Theta Reference Manual*, MIT Laboratory for Computer Science, Cambridge, MA, 1994.
- [20] M. D. McIlroy, *Mass Produced Software Components*, in P. Naur, Randell B. and Buxton, J.N., ed., *NATO Conference on Software Engineering, NATO Science Committee*, Petrocelli-Charter, Garmisch, Germany, 1968, pp. 88-98.
- [21] D. L. Parnas, *On the Criteria to be Used in Decomposing Systems into Modules*, *Communications of the ACM*, 15 (1972), pp. 1053-1058.
- [22] A. Schmolitzky, *Ein Modell zur Trennung von Vererbung und Typabstraktion in objektorientierten Sprachen (A Model for Separating Inheritance and Type Abstraction in Object Oriented Languages)*, *Ph.D. Thesis, Dept. of Computer Structures*, University of Ulm, Germany, 1999.
- [23] D. Schweisguth, *Second-generation aspect-oriented programming*, *Javaworld*, <http://www.javaworld.com/javaworld/jw-07-2004/jw-0705-aop-p3.html>, July, 2004.



---

## About the authors



**J. Leslie Keedy** retired from the position of Professor and Head, Department of Computer Structures, University of Ulm, Germany in 2005, where he previously led the Timor language design and the Speedos operating system design groups. His email address is [keedy@jlkeedy.net](mailto:keedy@jlkeedy.net). His biography can be visited at [http://www.jlkeedy.net/biography\\_short.php](http://www.jlkeedy.net/biography_short.php)



**Christian Heinlein** is Professor for Fundamentals of Computer Science and Software Engineering at Aalen University, Germany. In his research, he has developed "Advanced Procedural Programming Languages", which are both conceptually simpler and more flexible than standard object-oriented languages. More information about him and his work can be found at [www.htw-aalen.de/personal/christian.heinlein](http://www.htw-aalen.de/personal/christian.heinlein).



**Gisela Menger** received a Ph.D. in Computer Science from the University of Ulm in 2000. She recently retired from the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering.