References


Figure 6: The architecture of the GOM-system
\textbf{declare} increaseAge (\( \forall t_1 \leq [\text{Age::int}] \)): \( \forall t_1 \| \text{int} \rightarrow \text{void} \\
\textbf{code} \quad \text{increaseAgeCode};

The signature makes the operation \textit{increaseAge} applicable on any tuple structured instance that provides an \textit{Age} attribute of type \textit{int}. On the other hand, the operation

\begin{verbatim}
\textbf{declare} oldOne (\( \forall t_1 \leq \text{Student} \)): \( \forall t_1 \| \rightarrow \text{bool} \\
\textbf{code} \quad \text{oldOneCode};
\end{verbatim}

is only applicable on \textit{Student} instances—or instances of a subtype of \textit{Student}, e.g., \textit{PhDStudent}. The above declaration is actually equivalent to

\begin{verbatim}
\textbf{declare} oldOne: \text{Student} \| \rightarrow \text{bool} \\
\textbf{code} \quad \text{oldOneCode};
\end{verbatim}

because the substitutability of subtype instances is implicitly facilitated in GOM.

5 Conclusion

In this paper it was shown that one can achieve both, a maximum of flexibility and statically verifiable type safety—without loopholes—in an object-oriented data model. It was illustrated that inheritance alone, under the constraint of strong typing, restricts the expressiveness of the model in particular for operations based on collection types. Therefore, we developed a framework that unifies inheritance and bounded polymorphism. For this purpose an ordering on types was introduced that combines explicit (user-defined) subtype ordering for inheritance with an order based on the structural type representation for controlled polymorphism.

We have developed a type checking algorithm for GOM that is based on unification, similarly to the type checking algorithm devised by Milner for ML in [16].

The persistent object model GOM has been developed by utilizing the EXODUS “database generator” system that was realized at the University of Wisconsin [5]. The architecture of the GOM prototype is shown in figure 6. As can be seen, GOM is realized as a cross-compiler translating GOM types and operations into equivalent C structures and routines. The schema manager was—by utilizing a bootstrap strategy—already implemented as a GOM application. GOM will be used in several non-standard database applications from mechanical engineering, e.g., logistics control, shop floor planning, material flow simulation, etc.

Acknowledgement

P. C. Lockemann’s continuous support of our research is gratefully acknowledged. Our colleague C. Kilger carefully read a preliminary draft of this paper. Further, we would like to thank our students U. Degel, W. Häfelinger, A. Horder, D. Kossmann, K. Leberer, A. Lopes de Lima, U. Oetken, H. Ott, A. Papapostolou, K. Peithner, A. Saad, B. Sobottke, H. Spies, M. Steinbrunn, A. Tetzner, and R. Waurig for their help in “getting the prototype running”.

This work was partly supported by a grant from the Deutsche Forschungsgemeinschaft (DFG; German Research Council) within the interdisciplinary cooperation project SFB 346, sub-project A1 “Cooperation in Distributed Object Bases”.
\textbf{declare} translate\textsubscript{C}: CylinderSet \parallel\textbf{Vertex} \rightarrow \textbf{void code} translateCode\textsubscript{C};

is monomorphically defined for the receiver type \textit{CylinderSet}. This operation is not applicable on an object of type \textit{PipeSet}. Therefore, the operation would have to be reintroduced for \textit{PipeSet} as

\textbf{declare} translate\textsubscript{P}: PipeSet \parallel\textbf{Vertex} \rightarrow \textbf{void code} translateCode\textsubscript{P};

Exactly this problem, that no operations can be defined for more than one set type, will be mitigated by incorporating \textit{bounded polymorphism} [4] in our model. In addition, polymorphism extends the expressiveness of the language in such a way that polymorphic operations may be defined which are based on a specified minimal structure of the arguments; this makes it possible to apply the same operation on arguments which are not related by the explicit subtype hierarchy and, thus, extends the expressiveness beyond other well-known object models, e.g., Smalltalk and GemStone.

Our concept of bounded polymorphic operations allow the declaration and implementation of exactly one \textit{translate} operation for all set types containing element objects with a well-defined operation \textit{translate} of their own:

\textbf{polymorph declare} translate (\{t\textsubscript{1} \leq \{t\textsubscript{2}\}, \forall t \leq (\text{translate: Vertex} \rightarrow \text{void})): \forall t\parallel\textbf{Vertex} \rightarrow \textbf{void code} translateCode;\]

The \{t\textsubscript{1}\} and \{t\textsubscript{2}\} denote type variables. From the signature it follows that \textit{translate} is applicable on objects of all types that hold the type bounds specified within the parenthesis:

1. \{t\textsubscript{1}\} is a set type with elements of type \{t\textsubscript{2}\}

2. the element type \{t\textsubscript{2}\} has an appropriate \textit{translate} operation in its signature.

Although the operation notation of attributes—consisting of associated VCO and VTO pairs—can be used if a polymorphic operation requires the existence of a distinguished attribute, GOM offers a special notation for this. Consider an operation computing the average length of all elements of a set:

\textbf{polymorph declare} avgLength (\{t\textsubscript{1} \leq \{t\textsubscript{2}\}, \forall t \leq [\text{Length: float}]): \forall t\parallel \rightarrow \text{float code} avgLengthCode;\]

The declaration of an operation does not make any sense unless an appropriate implementation can be provided. Here is the polymorphic implementation of the \textit{translate} operation that also uses type variables to constrain local variables. As can been seen, this implementation works on all objects that own a \textit{translate} operation and, therefore, is a correct implementation for all types on which \textit{translate} is applicable.

\textbf{define} translateCode (t)
\textbf{var} element: \{t\textsubscript{2}\};
\textbf{begin}
\textbf{foreach} (element in self)
\quad element.translate (t);
\textbf{end define} translateCode;

We use polymorphic operations to define elementary built-in operations for set and list types like \textit{insert}, \textit{delete}, \textit{nextElement}, and so on in our prototype implementation of GOM.

Using polymorphic operations also facilitates to expand operations on tuple-structured types across the explicit type hierarchy. We revisit our \textit{Wine} and \textit{Student} example. There are some operations that may very well be suitable for instances of either kind, e.g., the operation \textit{IncreaseAge} which could thus be defined as:
3.5 Summary

Figure 5 summarizes the interdependences worked out in this section. As we have (intuitively) illustrated it is equivalent for a language $\mathcal{L}$ to be strongly typed or to fulfill the substitutability property. Further the substitutability property is equivalent to the property that any two types related by the subtype relationship define a legal refinement.

However, the reader should bear in mind that in GOM not every legal refinement—from the type safety point of view—is actually allowed. In order to avoid unintended inheritance substitutability is still restricted by the explicit user-defined supertype hierarchy, that is, only instances of explicit subtypes can be substituted for instances of a supertype. This way we avoid the problems—discussed in Section 2.3—concerning unintended subtyping of, e.g., Wine and Student.

4 Polymorphism

Inheritance on tuple-structured types provides a high degree of flexibility on the use of operations: all operations are automatically applicable on subtypes of $t$. Furthermore, refinement allows for specialized behavior of operations when applied to instances of a subtype.

Unfortunately, the outlawing of subtyping on collection types implies that the inheritance-based polymorphism is not applicable for collections, e.g., lists and sets. Strictly speaking, even such basic operations as insert and delete would have to be defined monomorphically for each individual set type:

\begin{verbatim}
declare insert_C: CylinderSet || Cylinder -> void code insertCodeC;
declare insert_P: PipeSet || Pipe -> void code insertCodeP;
\end{verbatim}

By overloading, the two operations could still have the same name, i.e., insert, because from the statically determined receiver type the “correct” version of insert could be deduced.

While such operations as insert and delete could automatically be generated by the system upon definition of a new set type the problem becomes worse when considering application-specific operations. For example, the operation translate for translating all elements of a set containing Cylinder objects that is declared as follows:
Σ_{AwardPaper} ≡ [[author: → FullProfessor, author: ← FullProfessor, ...]]

The value receiving operation author—of AwardPaper is not a legal refinement of the respective operation of Paper. Furthermore, the type signatures show that attributes must not be retyped by supertypes of the original attribute types either, e.g., typing the attribute author of AwardPaper with Person which is a supertype of Professor would lead to a type conflict, too.

### 3.4 Subtyping of Collection Types

Many people are tempted to view a collection type with element type $t_1$ as a subtype of collection type with elements of type $t_2$ if $t_1 \leq t_2$ holds. For instance, one is tempted to treat the type PipeSet as a subtype of CylinderSet (cf. Section 2.2.1), where PipeSet may be defined as follows:

```plaintext
type PipeSet is {Pipe};
```

We will now prove that the intuitive—and tempting—attempt of treating PipeSet as a (substitutable) subtype of CylinderSet is wrong. Suppose we have the following (persistent) variable declarations:

```plaintext
var ManyPipes: PipeSet;
var ManyCylinders: CylinderSet;
```

If PipeSet were a subtype of CylinderSet the assignment

```plaintext
ManyCylinders := ManyPipes;
```

would be valid and result in the sharing of the set object referred to by ManyPipes.

The dilemma is, that over the variable ManyCylinders, one could insert Cylinder instances into the set object of type PipeSet, e.g.:

```plaintext
ManyCylinders.insert(SomeCylinder);
```

This has the catastrophic side effect that the object to which ManyPipes and ManyCylinders refer is no longer properly typed.

Again, we will consider the type signatures to give a more formalized explanation of the relationship between strong typing and subtyping of collection types. The types CylinderSet and PipeSet have the following signatures if we assume that the operation insert is defined on sets:

```plaintext
Σ_{CylinderSet} ≡ [[insert: Cylinder → void]]
Σ_{PipeSet} ≡ [[insert: Pipe → void]]
```

The signatures of insert in the two type signatures are not valid refinements because the insert operation of PipeSet requires an instance $\leq$ Pipe whereas CylinderSet is “happy” dealing with instances $\leq$ Cylinder. Thus, PipeSet instances cannot safely be substituted for CylinderSet instances.

In summary of this section we can state that subtyping of collection types need not be forbidden generally; however, only collections with identical element types can possibly be valid subtypes of each other. Refining operations of a collection type in the appropriate way and adding new ones to a collection type leads to legal subtypes with compatible type signatures. But—to stress it again—the subtyping of collection types of different element types—even if one is the subtype of the other—is illegal.
As can be seen, FullProfessors are “better” than “normal” Professors since they are able to produce “better” output, i.e., AwardPapers, with “less good” input, i.e., “general” Students. Thus, whenever a “normal” Professor shall write a Paper one can substitute him by a FullProfessor without any loss.

Obviously, a language specifying types and the type ordering \( \leq \) fulfills the substitutability property if and only if the refinement property holds for any two types related by \( \leq \) within the language.

The refinement property has three important consequences for the subtyping rules in strongly typed object models:

- inheritance must follow the “inherit all” paradigm, that is, the subtype inherits all the attributes and operations of the supertype. It is—however—possible to refine some operations in accordance to the two refinement rules stated above. An example is the refinement of the inherited volume operation in Pipe or the refinement of the writePaper operation in FullProfessor.
- retyping of attributes must be forbidden
- collection types cannot be subtyped in the usual way by subtyping the element type

It is obvious that the inherit all paradigm is necessary to guarantee the refinement property. We illustrate the other two consequences by means of examples.

### 3.3 Retyping of Attributes is Illegal

Strong typing imposes a severe restriction on subtyping of tuple types. It is not allowed to redefine attribute types without violating type consistency. This will be shown by way of an example. Reconsider the type hierarchy of Figure 4 and let us sketch the two type definitions of Paper and AwardPaper:

```plaintext
type Paper is [author: Professor];  
type AwardPaper is [refine author: FullProfessor];  // ILLEGAL attribute refinement
```

As outlined in Section 3 all instances of type AwardPaper must be substitutable for all instances of type Paper wherever they appear in some program. Unfortunately, the example types above do not warrant this. Consider the following program fragment.

```plaintext
var p: Paper;  
  prof: Professor;  
  ap: AwardPaper  
...  
(1)  p.author := prof;  // okay  
(2)  p := ap;  // okay by substitutability  
(3)  p.author := prof;  // potential type violation
```

The principle of substitutability demands that the statement (2) is correct since the variable \( p \) can legally be associated with an AwardPaper instance. However, then the statement (3) may lead to a type violation since the variable \( prof \) could refer to an “ordinary” Professor which is then assigned as the author of the AwardPaper instance—a clear type violation.

More formally the contradiction of attribute retyping and strong typing can be expressed by giving the type signatures of both types.

\[
\Sigma_{Paper} \equiv \{ \text{author}: \rightarrow \text{Professor}, \text{author}: \leftarrow \text{Professor}, \ldots \} 
\]
Paper that are direct subtypes of root type ANY are specialized by FullProfessor, PhDStudent, and AwardPaper, respectively. However, we will not show the complete definition of these types in detail. Instead we will concentrate on one central operation whose declaration is as follows:

\[
\text{declare} \ \text{writePaper: Professor} \ || \ \text{PhDStudent} \rightarrow \text{Paper code writePaperCode;}
\]

Since we have FullProfessor \(\leq\) Professor the substitutability property states that any FullProfessor can write a paper with any PhDStudent.

Note, however, that the notion of substitutability is not concerned with the semantics of the operations. As long as the operations are compatible concerning their syntactical representation, i.e., name, input types, and result type, they are considered substitutable even if the respective operations are semantically totally different. Semantical compatibility is not accounted for in our typing framework.

In order to discuss the consequences of the substitutability requirement we define for every type \(t\) its signature \(\Sigma_t\). It contains all operations that are specified in the public clause. Remember that for attributes \(A_i : t_i\) defined for type \(t\) — or inherited from a supertype — none, one, or both of the two operations \(A_i := t_i\) and \(A_i := t_i\) may be made public. The last construct indicates that \(A_i\) is a value receiving operation and, hence, can only occur on the left hand side of an assignment. For our Cylinder type we then have (see Section 2.1.1):

\[
\Sigma_{Cylinder} \equiv \{ \text{Radius: } \rightarrow \text{float, Radius: } \leftarrow \text{float,} \\
\text{Diameter: } \leftarrow \text{float, Length: } \rightarrow \text{float,} \\
\text{Center1: } \rightarrow \text{Vertex, Center2: } \rightarrow \text{Vertex,} \\
\text{rotate: float, char } \rightarrow \text{void, translate: Vertex } \rightarrow \text{void,} \\
\text{scale: Vertex } \rightarrow \text{void, volume: } \rightarrow \text{float,} \\
\text{surface: } \rightarrow \text{float} \}
\]

The type signatures are — in the next subsection — used to define legal substitutability of an object type \(t_2\) for \(t_1\) based on comparing the type signatures \(\Sigma_{t_1}\) and \(\Sigma_{t_2}\).

3.2 Refinement of Operations

We now define the refinement property which is equivalent to the substitutability property. Two types \(t\) and \(t'\) with the signatures \(\Sigma_t\) and \(\Sigma_{t'}\) can legally be related as subtypes, i.e., \(t' \leq t\), if and only if for every value returning operation \(op: s_1, \ldots, s_n \rightarrow s_{n+1}\) in \(\Sigma_t\) there exists an operation \(op: s'_1, \ldots, s'_n \rightarrow s'_{n+1}\) in \(\Sigma_{t'}\) such that

1. \(s_i \leq s'_i\) for \(1 \leq i \leq n\) (i.e., the input parameter types must be supertypes)
2. \(s'_{n+1} \leq s_{n+1}\) (i.e., the result type must be a subtype)

and for every value receiving operation \(op: s_1, \ldots, s_n \leftarrow s_{n+1}\) in \(\Sigma_{t_1}\) there exists a value receiving operation \(op: s'_1, \ldots, s'_n \leftarrow s'_{n+1}\) in \(\Sigma_{t_2}\) such that

1. \(s_i \leq s'_i\) for \(1 \leq i \leq n + 1\) (all parameters can be viewed as input parameters)

An example for legal refinement is as follows. In our University it takes a Professor and a PhDStudent to write a Paper. However, if a FullProfessor is going to write down his or her thoughts only a “general” Student is necessary to create one that is even awarded (AwardPaper). Therefore, writePaper is being refined within type FullProfessor as follows:

\[
\text{refine writePaper: FullProfessor } || \ \text{Student } \rightarrow \text{AwardPaper code writeAwardPaper;}
\]
(programming or database) language provides in some sense the set of all possible types together with the $\leq$ relationship.

### 3.1 Substitutability

Consider the case where within a given language $\mathcal{L}$ the two types $t$ and $t'$ are related by any reflexive and transitive partial ordering such that $t' \leq t$. The *substitutability property* then states that wherever an object of type $t$ is legally placed in some expression, e.g., in an operation invocation, the replacement of it by an object of type $t'$ results again in a legal expression. We define *legal* as *well-typed* and require that the execution of a well-typed expression does not result in a type error.\(^5\) Consequently, we call a language specifying types and the $\leq$ relationship on types to guarantee the *substitutability property* if and only if

for any objects $o_i$ of type $t_i$, $o'_i$ of type $t'_i$ where $t'_i \leq t_i$ ($1 \leq i \leq n$), and for any $n$-ary operation $op$ the expression $o_1 . op(o_2, \ldots, o_n)$ is well-typed and the result is of type $t_{n+1}$ then the expression $o'_1 . op(o'_2, \ldots, o'_n)$ is well-typed and the result is of type $t'_{n+1}$ with $t'_{n+1} \leq t_{n+1}$.

Let us assume that we can derive types for any expression occurring in a program written in the language under consideration but we are unable to derive at compile time which objects are held in certain variables or returned by certain expressions, e.g., retrieve statements in a database language. Then it is easy to see that the substitutability property is equivalent to strong typing which is defined such that type safety can be guaranteed at compile time.

**Example**  Let us illustrate the principle of substitutability on an example that could be taken from a common situation at the “University of the Future” where papers can be written at the fingertips of their authors sitting at their workstations and invoking the appropriate type specific operation. Figure 4 shows a sample type hierarchy. The three major types Professor, Student, and

\(^5\) We do not consider illegal array boundaries, division by zero etc. as type errors
A sample extension of myCylinderSet is shown in Figure 3.

As mentioned in Section 2.2.1 myCylinderSet may contain not only Cylinder instances but also Pipe instances (e.g., id33 in Figure 3) because Pipe is a subtype of Cylinder. In the general case this is referred to as substitutability in object-oriented systems. From the user's point of view, substitutability means that an instance of a subtype can occur wherever an instance of the supertype is required. An instance of type Pipe, i.e., the one with OID id33 is included in the set myCylinderSet that requires elements of type Cylinder or subtypes thereof. Substitutability will be revisited under the light of strong typing in the subsequent section.

The example also indicates the need that each object carries its own type identification. For the element id3 of the CylinderSet instance myCylinderSet the volume operation as defined for Cylinder has to be executed. But, for the element id33 the refined volume operation of Pipe has to be dynamically bound. Dynamic binding, however, is only possible if the object instances "know" to which type they belong, such that—according to the actual type—the most specific operation can be selected.

Subtyping and refinement of operations together with dynamic binding of the most specialized operation is a powerful instrument in object-oriented models. In many object models, however, subtyping violates strong typing of the programs. Also, refinement of operations cannot be allowed in an arbitrary manner. Therefore, we have to reconsider both issues in the light of strong typing in the subsequent section.

3 Strong Typing

Strong typing ensures that all expressions can be verified type consistent at compile time. Contrary to strictly typed languages, like Pascal, strongly typed languages do not always facilitate the static determination of the types of expressions at compile time. An example for this was already given in the previous section. Looping through the CylinderSet instance with OID id1 may yield a Cylinder instance or a Pipe instance—or any other subtype of Cylinder. However, in any case—whether Cylinder or Pipe instance—the expressions inside the foreach loop can be verified type safe at compile time. For this it had to be assured at compile time that whatever element can legally be in a CylinderSet instance it has to provide an operation volume that returns a floating point number. This—intuitively—is exactly what strong typing assures.

Strong typing requires fewer run-time checks as opposed to dynamically typed models—as, e.g., Smalltalk—and, thus, provides for faster programs. Also, errors can be detected at an earlier stage in the design of an application. On the other hand, the source code of programs is often larger than without strong typing. To overcome this disadvantage, we have introduced polymorphic operations (cf. Section 4) into GOM.

The main goal of this section is to explore the possibilities of subtyping and inheritance in strongly typed object-oriented languages. In the first subsection we introduce the substitutability property which, as we will see, is equivalent with strong typing. We then discuss the consequences of the substitutability property by relating it to the refinement property via type signatures. The result is that it is equivalent whether a language fulfills the substitutability property or all the subtypes expressible in the language fulfill the refinement property. The refinement property has three immediate, subsequently discussed, consequences for subtyping and inheritance. The summary of these relationships is given in Figure 5, and discussed in Section 3.5. These consequences together with user-specified subtyping then also lead to the design of the subtype and inheritance specifications obeyed in GOM.

Since the argumentation in this section concerns strongly typed languages in general forget for a moment about GOM types and GOM subtyping. A type will be seen as any collection of objects all with "similar" properties. Further assume that the types are ordered by a partial ordering denoted by \( \leq \). For two types \( t_1 \) and \( t_2 \), \( t_1 \leq t_2 \) means that \( t_1 \) is a subtype of \( t_2 \). We require that the
persistent type Cylinder is ...

In an analogous way, variables can be declared as persistent

persistent var myFavoriteCylinder: Cylinder;

Objects must be “told” explicitly that they should become persistent by invoking a second implicitly defined operation persistent

myFavoriteCylinder.persistent

One may argue that this concept is very awkward for database applications where most objects will be persistent. This problem can be overcome by inserting the statement

self.persistent

in the initialization operation that is automatically invoked upon object instantiation. This causes the persistency of all instances of the respective type.

There exist several dependencies among persistency of types, objects, and variables. First, only objects of persistent types can be made persistent since the type information of an object is always needed. (Of course, there may be transient objects of persistent types.) Second, it may cause dangling references if persistent variables reference transient objects. In general, however, it is not forbidden to let persistent variables reference transient objects—it is the programmer’s responsibility to maintain the database in a consistent state. Furthermore, persistent object types can only have persistent supertypes.

2.6 Summary

Let us summarize the most important concepts that have been introduced in this section on the following program fragment.

persistent var myCylinderSet: CylinderSet;
var c: Cylinder;

foreach c in myCylinderSet    // assume myCylinderSet refers to id1
    TotalVolume := TotalVolume + c.volume;

Figure 3: A Collection Example
tribute or an atomic attribute within some tuple object. Figure 2 shows a sample database occurrence with one Cylinder instance (with the OID id$_3$) and two Vertex instances (with OIDs id$_{13}$ and id$_8$). Thus, the Cylinder instance is represented as the triple

\[ o = (id_3, \text{Cylinder}, v) \]

where \( v \) corresponds to the internal representation as shown in the rectangular box on the left-hand side of Figure 2. The Vertex instance with OID id$_8$ is associated with the attribute Center1 of the Cylinder with OID id$_3$, and the other Vertex instance id$_{13}$ is associated with the attribute Center2.

### 2.4.5 Variables

GOM variables are constrained to particular types. Like attributes of tuple-structured types, their type can be either atomic, i.e., if they are associated with a value of type integer, float, etc., or they are constrained to an object type. In the latter case variables contain—alogously to the object-valued attributes—the OID of the associated object or NULL if no object has been assigned. In this sense object-valued variables are similar to pointer types in conventional programming languages. However, dereferencing is implicit and not controlled by any syntactic construct in GOM.

Before a variable can be used it has to be declared. For example:

```python
var myFavoriteCylinder: Cylinder;
```

As discussed above, referring to an object-valued variable or attribute—like Center1 within the Cylinder instance id$_3$—should be identical to accessing an atomic attribute—like Radius. Let us assume that the above declared variable myFavoriteCylinder refers to the Cylinder instance with OID id$_3$. Then

```python
myFavoriteCylinder.Radius
```

returns the value of the Radius attribute, i.e., 8.5, and

```python
myFavoriteCylinder.Center1.X
```

returns the value of the X-coordinate of the Cylinder's Center1. The latter expression is a so-called path expression for which GOM provides particular index structures, called access support relations [11, 12] in order to optimize their evaluation in queries.

### 2.5 Persistence

One of the key issues of GOM is that objects created in the above cited manner are not only available in the scope of one program. Objects, their type definitions, and also variables that reference objects can be made persistent.

The intention to make a type definition persistent can be expressed by the keyword `persistent`, e.g.,
2.4 Objects

2.4.1 Distinction between Values and Objects

We agree with Goguen [9] that it is important to distinguish between data (values) and objects. A data value, like the integer number 59, is an immutable, unchanging data entity that can never be modified. In this respect the elements of atomic data types, like integer, boolean, real, char, etc. are fixed, eternal items in the system.

By contrast, objects are persistent but potentially mutable, meaning that they can change their (internal) state. For example, the Person object named “Mickey Mouse” may have an attribute Age which is associated with the number 59. On his birthday this attribute changes by assigning a new integer value 60 to it. However, note that the integer value 59 did not change its state; rather another of the (eternal) integer values was utilized to replace the 59. In this sense objects are mutable by reassigning some of their attributes. But it is important to point out that the modification, e.g. the aging of a Person, does not result in a new object.

2.4.2 Object Identity

A logical consequence of object mutability is the need for object identity that is independent of the internal state of an object. In GOM every object is associated with a system-wide unique object identifier (OID) which is generated upon instantiation (birth) of the object and remains invariant throughout its life time. The object identifiers are entirely invisible to the user. They are internally used—by the system—to maintain references to the respective objects. Having an unambiguous identity provides a natural way for “shared subobjects” because the same object can be referenced—via its OID—any number of times.

Thus, a GOM object can be viewed as a triple

\((\#, \text{type}, v)\)

where \(\#\) represents the object identifier, \(\text{type}\) the object type of which the object was instantiated, and \(v\) denotes the internal representation of the object.

By contrast, atomic values do not require an object identifier because they are immutable. Therefore, their value is sufficient to unambiguously identify them. Consequently, elements of the atomic data types integer, float, char, boolean, etc. are not associated with an OID.

2.4.3 Object Instantiation

Every object type—except for virtual types which exist in GOM but will not be treated in this presentation—contain an implicit (predefined) create operation. For a tuple-structured type the predefined create operation returns an object where all attributes are set to NULL. In order to initialize some (or all) attributes differently one can provide a customized initialization operation of the same name as the object type—as exhibited in the Cylinder example. In this example the operation Cylinder has three parameters: one float value (the Radius) and two Vertex instances (the Center1 and Center2) of the newly created Cylinder instance. The Length attribute is not passed as a parameter since it can be computed as the distance between Center1 and Center2. This initialiser is automatically invoked upon creation of a new object—in this case the predefined create operation has to be supplied with the corresponding parameter of the initialiser.

2.4.4 Object References

We concur with Beeri [3] that referencing and dereferencing of object-valued variables (or attributes) should be implicit\(^4\). There should be no difference in accessing, for instance, an object-valued at-

\(^4\)Beeri’s critique was aimed at the Extra [6] object model and its ref and own concepts.
define connect (otherPipe) is

end type Pipe;

We will use this example to describe informally our subtype and inheritance concept.

- If the supertype is a tuple type all attributes are inherited, e.g., `Pipe` inherits the attributes `Center1, Center2, Radius, and Length` of `Cylinder`. Additional attributes, like `InnerRadius` of `Pipe`, can be defined to specialize a subtype. Subtyping of collection types is discussed in Section 3.4.

- All operations are inherited, e.g., the operations `rotate, translate`, etc. of `Cylinder`, are also applicable on `Pipe` instances. Additional operations like `connect` of `Pipe` can be defined to express the special semantics of subtype instances.

- The `public` clause is inherited from the supertype and may be augmented by further, public operations. However, it is not allowed—under strong typing—to exclude an operation from the public clause that was included by some (direct or indirect) supertype.

- Operations inherited from the supertype can be `refined`, i.e., they can be reimplemented. This has been done with the `volume` operation of `Cylinder` in the example above. Note that in the refined implementation we actually utilized the inherited `volume` definition of `Cylinder` by invoking `super.volume`. This invocation returns the computed volume as if the respective instance were a `Cylinder`, i.e., it returns `self.Radius * self.Radius * 3.14 * self.Length`.

Unlike many type-theoretic approaches that try to model object-orientation with typed lambda calculus, e.g. FUN designed by Cardelli and Wegner [4], subtyping in GOM is explicitly controlled by the user-defined type hierarchy (see also [9] on this topic). In FUN—and some other approaches—subtyping is implicitly controlled by matching the object structures. Therefore, in FUN the two tuple types

```plaintext
type Wine is
  body [Name: string; Age: int;]
end type Wine;

type Student is
  body [Name: string; Age: int; GPA: float;
end type Student;
```

are related in such a way that `Student` is a subtype of `Wine` (since it provides all the attributes of `Wine` and some other). Therefore, any operation on a `Wine` instance is—by inheritance—also applicable on a `Student` instance which may lead to surprising results when, for example, the operation `BestDrinkingTemperature` is invoked on a `Student` instance. Therefore, we see a strong need for user-controlled subtyping which is achieved in GOM by restricting subtyping to the explicit, user-defined supertype-hierarchy.

Note that GOM—like all other well-known object-oriented models—coupled subtyping with inheritance. This sometimes leads to anomalies whenever the “inherits-all” semantics of inheritance does not conform to the semantics of the subtyping. An example is as follows: a `Cube` is certainly a specialization of a `Cuboid` and, therefore, should be defined as a subtype of `Cuboid`. However, the modeling of a `Cuboid` requires the attributes `Length, Height, and Width` whereas a `Cube` has only one attribute `Length`. In [13] we have devised a formalism, called `constraint inheritance`, that allows to restrict the inherited attributes and, thus, alleviates the problems induced by combining subtyping and inheritance.
Radius. As an abbreviation for both, the VCO and the VTO we could use \textit{Radius} which implicitly refers to \textit{Radius$\to$} and \textit{Radius$\leftarrow$}.

Aside from the predefined VCOs for attribute assignment it is also possible to define new, customized value receiving operations. An example based on our type \textit{Cylinder} should illustrate this:

\begin{verbatim}
declare Diameter: Cylinder \leftarrow float code DiameterCode;

define DiameterCode is
    var d: float;
    begin
        receive d;
        self.Radius := d/2;
    end define DiameterCode;
\end{verbatim}

This operation can then be invoked by

\begin{verbatim}
c.Diameter := 10.0;
\end{verbatim}

in order to assign 5.0 to the \textit{Radius} of the \textit{Cylinder} object referred to by \textit{c}.

\subsection{2.2.1 Collection Types}

Aside from tuple-structures the body of an object type can be defined as \textit{set} or \textit{list} which we will refer to as collection types if the difference between them, i.e., lists or sets, is not significant. A set is denoted as \{\textit{t}\} where \textit{t} is a type name. A set of this type may only contain elements of type \textit{t} or subtypes thereof (cf. Sections 2.3 and 3). As usual for sets no duplicate elements are allowed, i.e., no two elements with the same identity can be elements of one set. A list is denoted as \textit{< t \textit{.} \textit{>}}. Lists are analogous to sets except that an order is imposed upon the elements and, thus, duplicate elements become possible. We give an example of a set type:

\begin{verbatim}
type CylinderSet is
    body {Cylinder}
end type CylinderSet;
\end{verbatim}

Objects can, of course, be elements of more than one instance of an appropriate collection type (see also Figure 3 in Section 2.6).

\subsection{2.3 Subtyping and Inheritance}

The set of GOM types can be structured by the super/sub-type relationship where subtypes inherit the properties, i.e., attributes and operations, from their supertype. Currently, GOM supports only single inheritance. As an example we define a type \textit{Pipe} as subtype of \textit{Cylinder}.

\begin{verbatim}
type Pipe supertype Cylinder is
    public InnerRadius, connect \!! InnerRadius$\to$ and InnerRadius$\leftarrow$
    body
        [InnerRadius: float;]
    operations
        declare connect: Pipe \to Pipe;
        refine volume: \to float code pipeVolumeCode;
    implementation
        define pipeVolumeCode is
            return (super.volume -
                self.InnerRadius * self.InnerRadius * 3.14 * self.Length);
\end{verbatim}
that are usual in “normal” programming languages (e.g., assignment, conditional statements, loops, etc.). Due to lack of space we will not discuss this language in further detail. Of particular interest is the operation Cylinder, which is used to initialize newly generated instances of the respective type. The more detailed discussion of object instantiation and initialization is given in Section 2.4.3.

Declarations and implementations of operations need not necessarily take place inside the type definition frame. An equivalent way to introduce, e.g., the weight operation for the Cylinder type at an arbitrary location is the following:

\[
\text{\textbf{declare}} \ \text{weight: Cylinder} \ | \rightarrow \text{float} \ \text{code}\ \text{weightCodeForCylinder;}
\]

The receiver type—here Cylinder—is followed by the “|” sign to distinguish it from the remaining parameters, if any. This “stand-alone” operation definition is of particular importance if one wants to add an operation to an existing type definition. The code clause specifies the name under which the implementation of this operation is to be found. Since GOM allows overloading of operations it is often necessary to specify an implementation name different from the operation name. This may even be needed within the type definition frame for overloaded\(^2\) operations.

### 2.2 Value Returning and Value Receiving Operations

In GOM we distinguish between value returning (VTO) and value receiving operations (VCO). VTO operations return a value (or an object) upon invocation while VCOs receive a value typically for attribute assignment.

The signature of a VTO operation op is as follows,

\[
\text{\textbf{declare}} \ op: t_1|t_2, \ldots, t_n \rightarrow t_{n+1};
\]

where \(t_1\) is the receiver type, the \(t_2, \ldots, t_n\) are additional argument types, and \(t_{n+1}\) is the return type.

A VCO operation op has the signature

\[
\text{\textbf{declare}} \ op: t_1|t_2, \ldots, t_n \leftarrow t_{n+1};
\]

where, again, \(t_1\) is the receiver type, \(t_2, \ldots, t_n\) are argument types, and \(t_{n+1}\) is the type of the value (object) being received within the operation.

For a tuple structured type \(t\) with attributes \([A_1 : t_1, \ldots, A_m : t_m]\) the following VTO and VCO operations are implicitly provided for each attribute \(A_i\) for \(1 \leq i \leq m\):

\[
\begin{align*}
\text{\textbf{declare}} \ A_i : t | \rightarrow t_i; & \quad \text{\textit{value returning operation}} \\
\text{\textbf{declare}} \ A_i : t | \leftarrow t_i; & \quad \text{\textit{value receiving operation}}
\end{align*}
\]

The operations are predefined; but, nevertheless, it is the type designer’s choice whether they are made visible by including them in the public clause or not. It is remarkable that both operations have the same name—the compiler derives the correct version from the context of the invocation. For example, for an object \(o\) of corresponding type, the invocation\(^3\)

\[
o.A_i := o.A_j;
\]

contains one VCO invocation—namely \(o.A_i\)—and one VTO invocation—namely \(o.A_j\). Thus, the VCO is always invoked to the left of an assignment sign (\(:=\)).

In order to distinguish VCO and VTO operations of identical name in, for example, the public clause one suffixes their name by a “\(\rightarrow\)” or a “\(\leftarrow\)”.

\(^2\)GOM provides for overloading of operations which is—due to space limitations—not discussed in this paper.

\(^3\)Let us assume that \(A_i\) and \(A_j\) are of identical type.
type Cylinder is
  public Center1 →, Center2 →, create, rotate, scale, translate,
  volume, surface, Radius →, Radius ←, Length →
  body
  [Center1: Vertex;
   Center2: Vertex;
   Radius: float;
   Length: float]
  operations
    declare rotate: float, char → void;
    declare translate: Vertex → void;
    declare scale: Vertex → void;
    declare volume: → float;
    declare surface: → float;
    declare Cylinder: float, Vertex, Vertex → void;
  implementation
    define rotate (Angle, Axis) is
      ...
    define translate (t) is
      begin
        self.Center1.translate(t);
        self.Center2.translate(t);
        !! assuming that translate is defined on Vertex type
      end define
    define scale (s) is
      ...
    define volume is
    define surface is
      ...
    define Cylinder (r, c1, c2) is
      begin
        self.Radius := r;
        self.Center1 := c1;
        self.Center2 := c2;
        self.Length := c1.distance(c2) !! assuming distance is defined in Vertex
      end define
  end type Cylinder;

  Figure 1: The Type Definition Frame for Cylinder
bounded polymorphism which is described in Section 4. Section 5 concludes this paper by describing the status of the GOM project.

2 The Basics of GOM

In this section the basic concepts of the object-oriented data model GOM are described—on an intuitive, somewhat informal basis by way of examples. In essence, GOM provides all the compulsory features identified in the “Manifesto” [1] in one orthogonal syntactical framework.

2.1 Types

Objects incorporate their structural and behavioral description. Objects with similar properties, i.e., structure and behavior, are classified in types. A new type is introduced using the type definition frame which has the following syntactical form:

\[
\begin{align*}
\text{[persistent] type \hspace{1em} \text{(type name)}} & \hspace{1em} \text{[supertype (supertype name)] is} \\
\text{[public (operations list)]} & \\
\text{[body (type structure)]} & \\
\text{operations} & \\
\text{\hspace{1em} (operation signature);} & \\
\ldots & \\
\text{\hspace{1em} (operation signature);} & \\
\text{implementation} & \\
\text{\hspace{1em} (operation implementation);} & \\
\ldots & \\
\text{end type (type name)};
\end{align*}
\]

A newly defined type has a unique \text{(type name)} which must not have been assigned before. The optional keyword \text{persistent} initiates that the type definition is being stored in the database schema. Currently, GOM supports single inheritance, thus the (optional) \text{supertype} clause may specify the one \text{(super type)}—if no supertype clause is specified the supertype \text{ANY} is implicitly assumed. The \text{public} clause lists all the type associated operations that constitute the interface of the newly defined type. The \text{body} clause precedes the definition of the structural representation of the type. We distinguish tuple-structured types (cf. Section 2.1.1) and collection types (cf. Section 2.2.1). The behavior of objects \text{(of a type)} is specified by a set of type-associated operations. The \text{operations} clause contains the abstract signatures consisting of a type-wide unique operation name, a list of input parameter types and the result type of the operations that are associated with the type. The implementation of these operations is supplied under the \text{implementation} clause.

2.1.1 Tuple-Structured Types: An Example

Rather than specifying the precise semantics of GOM type definitions let us illustrate the concepts by way of examples. A tuple consists of a collection of typed attributes. The tuple constructor is denoted as \([a_1 : t_1, \ldots, a_n : t_n]\) for pairwise distinct attribute names \(a_i\) and (not necessarily distinct) type names \(t_i\). In Figure 1 our running example type \text{Cylinder} is introduced as a tuple-structured type with associated, type-specific operations.

Declaration and implementation of the operations are provided in separate sections of the type definition frame. Operations can be implemented in a C-like syntax offering all control constructs

\footnote{Albeit the design of GOM was carried out before the “Manifesto” was written.}
and hierarchical model) and their associated database programming languages, e.g., Pascal R [17],
uniformly provide static type specificity. The recent emergence of object-oriented data models as
the (presumably) next-generation DBMS has led to a relaxation of static typing in favor of increased
flexibility and expressiveness. Unfortunately, in most newly developed object models the increase
in flexibility had to be paid for dearly: database operations could no longer be guaranteed type
safe. In order to achieve a high degree of reusability of built-in database operations and user-defined
operations type constraints were either completely or partially abolished for the sake of flexibility:

- In some data models no type information is kept. Objects can be freely created using the
  built-in type constructors. This approach is typically termed *loose typing*. This approach has
  its roots in the programming language LISP; one representative data model adhering to this
typing philosophy is FAD [2].

- The *dynamically typed* models let the database designer specify the outer level of the objects;
  but the components (attributes, set elements, etc.) are untyped, i.e., they may refer to any
  object. The precursor of this approach in the programming language area is Smalltalk-80 [10],
  which gave rise to some developments in the database area, e.g., GemStone [7] and—to some
  extent—Orion [14].

The two above mentioned classes of object-oriented data models cannot guarantee type safety
of database operations at compile time. We argue that the lack of type safety in object bases
constitutes a more severe problem than in object-oriented programming languages: an object base
is a highly shared, persistent resource which is modified by a variety of more or less knowledgeable
users. Furthermore, many database facilities, e.g., access support, concurrency control, recovery,
etc., are inherently more difficult and less efficient and robust under dynamic typing.

Therefore, a third class of object models was designed along the lines of Simula 67 [8] in order to
try to reconcile type safety and object-oriented features, such as subtyping, inheritance, operation
overriding, and late binding. This approach is typically called *strong typing*: all expressions in the
language can be verified type consistent at compile time even though the exact type cannot always be
determined statically. However, in all models that we know of there are either some loopholes where
dynamic type checking is still required or, the constraints imposed by strong typing severely reduce
flexibility. An example of the former is O₂ [15], in which the designers consciously incorporated some
features that prevent (complete) static type verification for the sake of expressiveness. These features
include (cf. section 3):

- retyping of attributes in a subtype (this, in general, violates strong typing even if the new type
  is a subtype of the original one)

- exceptional attributes (attributes that are only present in some instances of the type, but not
  in all)

- subtyping on set- and list-structured object types

Our main contribution in designing GOM is the development of a framework which provides for (complete)
static type verification while—at the same time—the flexibility and expressiveness associated
with the object-oriented paradigm is retained.

The remainder of this paper is organized as follows. In Section 2 the basic features of the GOM
model are described by way of examples: object identity, object sharing, instantiation, subtyping
and inheritance, type-associated operations, operation refinement (overriding), and late binding. In
Section 3 we discuss the concepts that provide for strong typing. In particular, we identify some
commonly encountered loopholes that violate strong typing and describe the way we avoid these
problems in GOM. In order to achieve type safety we had, to some degree, sacrifice some flexibility
that comes for free in dynamically typed models. This flexibility is regained in GOM by incorporating
GOM:
A Strongly Typed Persistent Object Model
With Polymorphism

A. Kemper    G. Moerkotte    H-D. Walter    A. Zachmann

Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe
Netmail: kemper|moer|walter|zachmann@ira.uka.de

Abstract

In this paper the persistent object model GOM is described. GOM is an object-oriented
data model that provides the most essential object features in a “lean” and coherent syntactical
framework. These features include: object identity, objectinstantiation, subtyping and
inheritance, operation refinement, dynamic (late) binding. One of the main goals in the design
of GOM was type safety. In order to achieve this we developed a strongly typed language that
enables the verification of type safety at compile time. It is shown in this paper how commonly
encountered “traps” for strong typing are avoided in GOM by specifying a very clean subtyping
semantics on the basis of substitutability and type signatures.

The typing rules that enforce strong typing at compile time somewhat restrain the flexibility
of the object model because subtyping and inheritance has to be restricted—in particular—for
collection-valued types. The solution to regain the expressiveness that was traded off for safety is
by combining inheritance and explicitly controlled operation polymorphism. To make operations
polymorphic signatures may contain type variables which are substituted by named types at
compile time.

1 Introduction

The object-oriented database area suffers from the lack of a commonly agreed upon object model.
Some attempts have been made to identify the most salient features that a database model has to
provide in order to be considered object-oriented. The “Manifesto” [1] distinguishes among compul-
sory features and optional features that have to be incorporated in object-oriented DBMSs. The
object model GOM that is discussed in this presentation was designed with the intention to provide
a “research vehicle” for investigating object-oriented database systems. As such, GOM incorporates
the essential constructs that have emerged in the past decade of work in object-oriented databases
in one coherent—yet syntactically lean—framework. By adhering to commonly agreed upon object
concepts we hope that our work on particular subjects, e.g., typing and optimization, will be applica-
table to a broad range of other object models. This should help to overcome the problems of “research
transfer” that are due to the diversity of existing object models. Among the features GOM provides
are: object identity, object sharing, instantiation, subtyping and (single) inheritance, type-associated
operations, operation refinement (overriding), and late dynamic binding of refined operations.

One of the major goals in designing GOM was type safety. Until recently, typing was not a
“hot” topic in database research: all conventional data models (relational, CODASYL network,