Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract. The basic idea of open types is to separate the definition of types from the definition of their constituents, i. e., their base types (or superclasses to use object-oriented terminology) and their data members (or fields). This is in complete contrast to traditional record types and object-oriented classes, which are *closed* in the sense that the set of their constituents is fixed once the type has been defined. It will be shown, however, that this alternative approach opens the door to greatly enhanced expressiveness and increased flexibility. Even though the concept of open types is presented in this paper as a language extension for C++, the basic principles are actually language-independent and could be incorporated into any imperative programming language.

1 Open Types

1.1 Type and Attribute Definitions

An *open type* is defined by declaring its name with the keyword `typename`, e. g.:

```
typename Person;
typename Car;
```

Afterwards, a *single-valued attribute* such as `name` − corresponding to a data field in record notion − can be defined by declaring it as a kind of mapping from `Person`s to `string`s:

```
Person -> string name;
```

Here, the right hand side of the definition (`string name;`) looks identical to a C++ (member) variable definition.

Similarly, a *multi-valued attribute* such as `gnames` (given names) − corresponding to a data field whose type is an array or container type − is defined by using a double instead of a single arrow to indicate the multi-valuedness:

```
Person ->> string gnames;
```

1.2 Constructors and Mutators

To create, initialize, and modify objects of an open type `T`, the following *constructors* and *mutators* are provided.

The *parameterless constructor* `T()`, which might either be called explicitly or is called implicitly for variables of type `T` which are not initialized explicitly [8], returns the *null object* of type `T`, i. e., actually *no object*. In contrast, the *attribute-initialization constructor* `T(@attr, val)` creates a distinct *new object* of type `T`, i. e., an object that is different from null and any other object, and initializes its attribute `attr` with value `val`.

Similarly, the *attribute mutator* `obj(@attr, val)` sets the value of attribute `attr` of object `obj` to `val` (if `attr` is a single-valued attribute) or adds `val` to `obj`'s values of attribute `attr` (if `attr` is a multi-valued attribute) and returns the object `obj`. This allows straightforward combinations of a constructor call with one or more mutator calls to create an object with multiple initial attribute values, e. g.:

```
Person p = Person(@name, "Hoare")(@gnames, "Charles")
 (@gnames, "Anthony")(@gnames, "Richard");
```

Here, the constructor call `Person(@name, "Hoare")` creates a new `Person` object, initializes its attribute `name` with the string `"Hoare"`, and returns the object. This object is directly used in the mutator call `...(@gnames, "Charles")` which initializes its attribute `gnames` with `"Charles"` and returns the same object. This is again used in and returned by the subsequent mutator calls `...(@gnames, "Anthony")` and `...(@gnames, "Richard")` which in turn add the strings `"Anthony"` and `"Richard"` to the values of attribute `gnames`. Finally, the object returned by the last mutator call is assigned to the `Person` variable `p`.

To create an *empty object*, i. e., a distinct object which is different from null and any other object, but does not possess any attribute values yet, the *Boolean constructor* `T(flag)` can be used. If the Boolean value `flag` is `true`, a unique empty object is created, while otherwise a null object is returned, i. e., `T(false)` is equivalent to just `T()`. On the other hand, a call `T(@attr, val)` to the attribute-initialization constructor is actually just a shorthand for `T(true)(@attr, val)`, i. e., a call to the Boolean constructor followed by an appropriate mutator call.

In addition to these predefined constructors of open types, it is possible to define arbitrary *user-defined constructors*, e. g.:

```
// Create person with given name g and name n.
Person (string g, string n) {
  return Person(@name, n)(@gnames, g);
}
```

In contrast to normal C++ constructors (and constructors in other object-oriented programming languages) which must be defined (or at least declared) inside their class and must not explicitly return anything, but rather initialize the implicitly available object `this`, user-defined constructors of open types are much like ordinary (global) functions whose result type and name coincide (and therefore only one of them is specified in their definition). In particular, there is no implicitly available object `this`, and the constructor must explicitly (create and) return an object, typically by calling one of the predefined constructors. Furthermore, just like attributes, constructors can be defined *successively* on demand.

1.3 The Attribute Inspection Operator @

To inspect the attribute values of a given object, the *attribute inspection operator* `@` can be used, quite similar to the way the dot operator is used to access class members in C++ and other languages, e. g.:

```
string n = p@name;
```

For a single-valued attribute such as `name`, its current value is returned, i. e., the value that has been set for this attribute by the most recent mutator (or attribute-initialization constructor) call for this object. If none of these operations has been executed for the object yet, i. e., the attribute does not possess any value, a well-defined *default value* is returned that is obtained by calling the parameterless *default constructor* of the attribute's type (i. e., `string` in the current example). Ideally, this constructor should return *null* to indicate the absence of any real value [2], but in principle any value (e. g., an empty string or zero for numeric types) is acceptable.

If a multi-valued attribute such as `gnames` is inspected with the `@` operator, the values added to this attribute by all mutator (and attribute-initialization constructor) calls performed for this object so far are returned as an *ordered sequence*. Even though it is possible to grasp such a sequence as a whole, it is typically processed element by element using a tailored iteration statement, e. g.:

```
for (string g : p@gnames) cout << g << " ";
```

This prints `p`'s given names in the order in which they have been added, i. e., `Charles Anthony Richard`. Alternatively, it is possible to directly inspect individual values of such a sequence by applying the well-known index operator, e. g.:

```
string g2 = p@gnames[2];
```

to obtain the second given name of `p`, i. e., `"Anthony"`. Similarly to inspecting the value of a non-existent single-valued attribute, inspecting a non-existent value of a multi-valued attribute by using an out-of-range index yields a well-defined default value that is obtained in the same way as described above. Therefore, expressions such as `p@gnames[0]` or `p@gnames[4]` will return a null string in the current example.

It should be noted that the attribute inspection operator always returns an *R-value* [8], i. e., a value which must not occur on the left hand side of an assignment operator. Therefore, attribute update operations must only be performed by mutator calls, not directly by assignments such as:

```
p@name = "Hoare"; // Syntax error!
```

1.4 Inspecting and Modifying Null Objects

Trying to inspect or modify a member of "object null," i. e., the "object" referenced by a null pointer, is illegal in C++ and many other languages and usually leads to a run

time error such as a `SIGSEGV` (segmentation violation) signal or a `NullPointerException`, since a null pointer actually does not refer to any object.

In contrast to that, inspecting and modifying attributes of open types is well-defined even for null objects: While inspecting such an attribute is equivalent to inspecting a non-existent attribute, i. e., returns the attribute's default value, modifying such an attribute simply has no effect. The main reason for these unusual definitions is convenience, since they allow to omit many otherwise necessary checks. To test, for example, whether `p`'s name is `"Hoare"`, one can simply write `if (p@name == "Hoare")` − instead of `if (p && p@name == "Hoare")` − even if `p` might be null; if it actually is, `p@name` is null, too, and therefore, as expected, different from `"Hoare"`. Simultaneously, programs tend to become more robust since inadvertently omitted checks will not lead to run time errors, but usually merely to unsatisfied conditions.

Similarly, the definition that mutator calls on null objects are silently ignored frequently reduces the need to explicitly distinguish between real and null objects, and again, inadvertently omitting such distinctions does not lead to run time errors (cf. [2]).

1.5 Object Deletion

In contrast to normal C++ objects, which must be explicitly deleted by the programmer to reclaim their storage, objects of open types are automatically *garbage-collected* when they have become unreachable, quite similar to objects of classes in Java, Eiffel, Smalltalk, and many other programming languages.

In addition to and independently from this automatic storage reclamation, it is also possible to explicitly *delete* objects − even while they are still referenced. Although this might appear strange at first sight, there are reasonable practical examples where this is useful. If, for instance, a car has been scrapped, it does not exist anymore, even though it might still appear in the list of all cars of its (previous) owner.

In contrast to C++, however, where the deletion of an object might lead to dangerous dangling pointers, deletion of an open type object causes all remaining references to the object to become null immediately and automatically. By that means, it is always possible to reliably detect that an object has been deleted. Furthermore, since null objects can be safely inspected and modified, too, neither run time errors nor undefined behaviour will occur if deleted objects are used without care. Since object deletions might be performed unexpectedly, this is another strong argument for the definitions given in Sec. 1.4.

2 Bidirectional Relationships

Basically, a *bidirectional relationship* between two types is also a kind of mapping from one type to the other, with the additional possibility to directly access the *inverse mapping*. Since both of these mappings might be either single- or multiple-valued, there are four different kinds of relationships altogether, one to one, one to many, many to one, and many to many, expressed by corresponding bidirectional arrow symbols `<->`, `<->>`, `<<->`, and `<<->>`, respectively. Furthermore, there are two special

kinds, i. e., *symmetric* one-to-one and many-to-many relationships, where the inverse mapping is equivalent to the original mapping.

For example, a one-to-many relationship between `Person` and `Car` called `cars` resp. `owner` can be defined as follows:

```
Person owner <–>> Car cars;
```

Reading this from left to right (and omitting the name on the LHS) yields a multi-valued attribute of type `Person`:

```
Person –>> Car cars;
```

while reading from right to left (and omitting the name on the RHS) yields a single-valued attribute of type `Car`:

```
Car –> Person owner;
```

representing the inverse mapping. However, only by combining both attribute definitions into a single relationship definition as shown above, they are actually treated as mutually inverse mappings, which means that a call to one of the mutators automatically implies a corresponding call to the other mutator with reversed roles.

For example, a mutator call such as `p(@cars, c)` adding `c` to `p`'s sequence of cars, implies the call `c(@owner, p)` assigning `p` as `c`'s owner, and vice versa. Furthermore, if `c` already possesses another owner `q` when either such call is made, `c` is first removed from the sequence of `q`'s cars.

3 Anonymous and Automatic Attributes and Relationships

3.1 Basic Principles

If the name of a single-valued attribute is omitted, it implicitly possesses the name of its target type, e. g.:

```
typename Address;
Person –> Address;

Person p = Person(@Adress, Address(...));
Address a = p@Adress;
```

Similarly, it is possible to omit one or both names of a bidirectional relationship.

If the arrow in an attribute declaration is followed by an exclamation mark, the attribute might be applied *automatically* on demand to perform an *implicit type conversion* from its source type (left of the arrow) to its target type (right of the arrow), e. g.:

```
Person –>! int pid;
```

This declares an `int` attribute `pid` of type `Person` which can be used just like any other attribute, with the additional property that an expression of type `Person` is implicitly convertible to an `int` value by automatically applying this attribute.

Similarly, it is possible to declare automatic relationships by adding an exclamation mark before or after the bidirectional arrow, depending on which direction of the relationship should be automatically applicable.

3.2 Modeling Type Hierarchies

Automatic one-to-one relationships can be exploited to model object-oriented type hierarchies with subtype polymorphism without requiring any additional mechanisms. For example, a new type `Student` (with a regular attribute `number` denoting the matriculation number) might be defined as a "subtype" of `Person` by declaring a one-to-one relationship between these types that is automatically applicable from the derived type to the base type:

```
// Declare Student as a "subtype" of Person.
typename Student;
Student -> string number;
Student <->! Person;
```

Typically, but not necessarily, such "subtype" relationships are anonymous.

A typical constructor for `Student` might be defined as follows:

```
// Create student with given name g, name n,
// and matriculation number m.
Student (string g, string n, string m) {
  // Create person subobject.
  Person p = Person(@name, n)(@gnames, g);

  // Create and return student object connected with p.
  return Student(@Person, p)(@number, m);
}
```

Now, a student named Peter Clark with matriculation number 777 can be created and used as follows:

```
// Create student.
Student s = Student("Peter", "Clark", 777);

// Print name and matriculation number.
cout << "Name: " << s@name << endl;
cout << "Number: " << s@number << endl;
```

Because the relationship between `Student` and `Person` is applied automatically on demand, the subexpression `s@name` is actually replaced by `s@Person@name`. Furthermore, all functions accepting `Person` arguments can be called with `Student` objects, too, and finally, a `Student` object can be used polymorphically as a `Person` object.

The fact that the relationship between `Student` and `Person` is bidirectional can be exploited to check whether a given `Person` object "is" actually a student (i. e., to perform a *dynamic type test*) and to access its student attributes if appropriate (i. e., to perform a *downcast*):

```
// Polymorphically use a student as a person.
Person p = Student("Peter", "Clark", 777);

// Check whether p is actually a student s ...
if (Student s = p@Student) {
  // ... and access its matriculation number.
  cout << "Number: " << s@number << endl;
}
```

This corresponds roughly to a `dynamic_cast` in C++ which returns a valid pointer to an object of a derived class if the cast has been successful and a null pointer otherwise.

By employing automatic relationships to model object-oriented type hierarchies, the traditionally distinct or even conflicting concepts of *aggregation* and *inheritance* have been merged into a single coherent concept. Furthermore, the fact that relationships can be defined incrementally, allows "supertypes" of a type to be declared later on, e. g.:

```
// Declare Vehicle as a "supertype" of Car.
typename Vehicle;
Car <->! Vehicle;
```

Despite its practical usefulness, such a possibility is missing in most object-oriented programming languages.

3.3 Multiple Inheritance

Of course, it is possible to use automatic relationships to model type hierarchies with multiple inheritance, too. For example, one might define a type `EmployedStudent` that is derived from both `Student` and another type `Employee`:

```
// Declare Employee as a subtype of Person.
typename Employee;
Employee <->! Person;

// Attributes and constructors of Employee.
Employee -> string company;
Employee (......) { ...... }

// Declare EmployedStudent as a
// subtype of Student and Employee.
typename EmployedStudent;
EmployedStudent <->! Student;
EmployedStudent <->! Employee;
```

Since both of these types in turn "inherit" from `Person`, the typical question arises whether an `EmployedStudent` object should possess one or two `Person` subobjects, i. e., whether `Person` is, in C++ terminology, a *virtual* base type or not. In C++, the corresponding decision must be taken when the types `Student` and `Employee` are

defined, even though it does not make any difference for *these* types. Therefore, it would be much more logical to answer the question when `EmployedStudent` is defined, because only for this type (and possible subtypes of it) the distinction is relevant. However, the concept of automatic relationships does not provide a way to specify the difference at the level of *declarations*: The four `<->!` relationships between the types `Person`, `Student`, `Employee`, and `EmployedStudent` merely specify that there are two ways to convert an `EmployedStudent` to a `Person`, either via `Student` or via `Employee`, but they do not specify whether these ways lead to the same destination, i. e., to the same `Person` object, or not. Even though this appears to be disadvantageous at first sight, it will turn out to be the most flexible approach possible.

To actually distinguish between virtual and non-virtual inheritance, one simply creates either one or two `Person` "subobjects" when creating an `EmployedStudent` object in a constructor, e. g.:

```
// Create employed student with given name g, name n,
// matriculation number m, and company c.
EmployedStudent (string g, string n, string m, string c) {
  Person p = Person(@name, n)(@gnames, g);
  Student s = Student(@Person, p)(@number, m);
  Employee e = Employee(@Person, p)(@company, c);
  return EmployedStudent(@Student, s)(@Employee, e);
}
```

Here, a *single* `Person` object `p` is created that is passed to both the `Student` and `Employee` constructors to create `Student` and `Employee` objects `s` and `e`, respectively, which *share* the subobject `p`. Afterwards, an `EmployedStudent` object with subobjects `s` and `e` is created and returned. Therefore, converting an `EmployedStudent` object created by this constructor to type `Person` always yields the same `Person` subobject, no matter whether the conversion is done via `Student` or via `Employee`.

3.4 Dynamic Object Evolution

The fact that an object of a derived type such as `Student` or `EmployedStudent` is actually a network of interconnected subobjects − even though this remains invisible except when constructing the objects −, can be exploited in a straightforward manner to implement *dynamic object evolution*. For example, it is almost trivial to transform an object that has been initially created as a bare person into a student, an employee, or even an employed student later, by simply creating additional associated subobjects, e. g.:

```
// Create a person object p.
Person p = Person("Peter", "Clark");

// "Transform" p to a student.
p(@Student, Student(@number, 777));
```

Conversely, it is also possible to delete subobjects to transform a specialized object to

a more general one, e. g.:

```
// "Transform" p back to a person.
delete p@Student;
```

Here, it does not matter whether `p` has been originally created as a `Person` or a `Student` (or something else).

By explicitly deleting the student subobject associated with `p`, all "student references" to this person automatically become null. Otherwise, if only the association between `p` and its student object would have been cut, these references would remain valid, but refer to a degenerate student object that does not possess an associated person object anymore. (According to the rules given in Sec. 1.4, accesses to this person object and its attributes would still be well-defined, however.)

It is even possible to create "hybrid" objects, such as a person that is both a student and an employee, even if no common "subtype" of these types (such as `EmployedStudent`) would exist.

4 Related Work

Aspect-oriented programming languages such as AspectJ [4] or AspectC++ [7] provide so-called *inter-type member declarations* or *introductions* to retroactively extend existing data structure definitions without needing to change the code of the original definitions. Nevertheless, some kind of recompilation or "weaving" is required by all these approaches: While the AspectC++ compiler needs the source code of the original definition together with all extension code to produce a new definition that is actually compiled by a C++ compiler, the AspectJ compiler is able to perform the combination on the byte code level. Frameworks such as JMangler [5] are even able to delay the final composition until load time, but in either case a class remains fixed once it has been loaded. Open types, on the other hand, allow attributes to be loaded dynamically, even for types which have already been instantiated.

Actually, aspect-oriented approaches still adhere to the traditional concept of records as fixed data structures and only make their definition more flexible, while open types support truly flexible objects whose storage size might vary over time.

The *Common Lisp Object System* (CLOS) [3] (and other languages based on similar ideas) deviate from the typical object-oriented approach that everything belonging to a class must be defined (or at least declared) in the class, by allowing methods (of so-called *generic functions*) to be defined separately and incrementally. However, the set of data fields making up a class must still be defined at once and cannot be extended later, except by redefining the whole class. In contrast, open types apply the "generic function principle," i. e., the possibility to define methods separately and independently, to data fields, too.

Description logic systems such as Classic [1] and Loom [6] provide data models which are very similar in nature to open types and have in fact influenced some of their ideas. They provide *concepts* (corresponding to open types) and *roles* (corre-

sponding to attributes and relationships), which are defined separately and independently, and roles might possess *inverse roles*. Furthermore, a running system can be extended by new definitions at any time.

However, since description logic systems are actually AI tools, providing powerful reasoning capabilities such as subsumption checking, automatic instance classification, and truth maintenance, using them as bare data models of a programming language would mean to break a fly upon the wheel. Therefore, open types might be viewed as the result of reducing a description logic system to a simple data representation system by stripping off all AI functionality.

References

[1] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick: "Living with CLASSIC: When and How to Use a KL-ONE-Like Language." In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 401−456.

[2] C. Heinlein: "Null Values in Programming Languages." In: H. R. Arabnia (ed.): *Proc. Int. Conf. on Programming Languages and Compilers (PLC'05)* (Las Vegas, NV, June 2005), 123−129.

[3] S. E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 − Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327−353.

[5] G. Kniesel, P. Costanza, M. Austermann: "JMangler − A Powerful Back-End for Aspect-Oriented Programming." In: R. E. Filman, T. Elrad, S. Clarke, M. Aksit (eds.): *Aspect-Oriented Software Development*. Pearson International, 2004, 311−342.

[6] R. MacGregor: "The Evolving Technology of Classification-Based Knowledge Representation Systems." In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 385−400.

[7] O. Spinczyk, A. Gal, W. Schröder-Preikschat: "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53−60.

[8] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.