Efficient Implementation of Open Types

Christian Heinlein, Fakultät für Informatik, Universität Ulm

Introduction

Open types have been presented earlier as an alternative data model for imperative programming languages that overcomes several limitations of traditional record- and class-based models. The basic idea is to *separate* type definitions from the definition of their attributes in order to allow *incremental definitions* of the latter. This implies that attributes are generally *optional*, i. e., individual objects of a particular type need not possess values for all attributes of the type. If a non-existent attribute is accessed, a well-defined *null value* representing the absence of any real value is returned.

Figure 1 shows a simple example of an open type `Expr` possessing various attributes for the representation of arithmetic expressions. Furthermore, three user-defined constructors to create objects of different kinds and three branches of a *global virtual function* `eval` to inspect objects are shown. For more details, the reader is referred to [1].

Attribute Management

Figure 2 illustrates the basic approach to implement open type objects possessing values for different subsets of attributes. Here, an open type value is actually a pointer to an *intermediate cell*, which in turn references an associated *data area* containing the object's attribute values (in decreasing order of their alignment restrictions to avoid any padding in between) and a shared *object descriptor* describing its contents. For each attribute of the open type (`Expr` in this example), an object descriptor contains the information whether and where the corresponding attribute value is stored in an object's data area: If the "slot" corresponding to the attribute contains a number, it denotes the relative address of the attribute's value in the data area, while an empty gray slot indicates the absence of the attribute.

Based on this information, an attribute access operation proceeds as follows: A unique ordinal number assigned to the attribute (e. g., `1` for `op`) is used to access the corresponding slot of the object's descriptor, and the offset value found there is used to access the attribute's value in the object's data area. If no offset value is found, however, reading the attribute simply returns the null value of its target type, while a write access must extend the object's data area in order to obtain space for the new attribute. (This might change the address of the data area, but the address of the intermediate cell stored in open type values remains stable.) Furthermore, the object's descriptor pointer must be redirected to the "successor" descriptor that contains exactly the same attributes as the current one plus the new attribute.

Since object descriptors are created dynamically on demand, this extended descriptor might not yet exist. In that case, it is created and inserted into the type's *descriptor repository*. Furthermore, it is compared with all currently existing descriptors in order to find its "successors," i. e., those descriptors containing the same subset of attributes plus one extra attribute, as well as its "predecessors," i. e., those descriptors for which the new one is a successor. These predecessor/successor relationships are remembered by placing a reference to the successor (indicated by an arrow in Fig. 2) in the predecessor's slot corresponding to the extra attribute. Using this information, an already existing successor descriptor can be found easily and efficiently later.

Typically, an object has to be extended a few times during its initialization, while afterwards the set of its attributes (but, of course, not necessarily their values) is expected to remain rather stable. Similarly, the descriptor "network" of an open type is expected to change a number of times during the initialization phase of a program, until the object descriptors and their predecessor/successor relationships required for the typical object initialization patterns of the program have been built up. Afterwards, most object descriptors needed to perform an object extension will be found immediately via these relationships.

If an attribute of a particular type is loaded dynamically (e. g., `cache` which is not shown in Fig. 1), all object descriptors belonging to this type must be extended by a new slot corresponding to this attribute. This can be achieved by iterating through the descriptor repository and reallocating each descriptor. To make sure that the descriptor pointers of objects remain stable in that case, an additional level of indirection is used that is not shown in Fig. 2. Since loading new attributes is not expected to happen very frequently, the effort for these reallocations is acceptable. In order to reduce it even further, descriptors are always allocated with a certain number of extra slots in advance (indicated by the dashed boxes in Fig. 2).

Performance Results

To compare the run time efficiency of the open type implementation with standard class-based object-oriented systems, a simple test program creating, inspecting, and manipulating randomly structured object graphs has been written in standard C++, Java, and C+++ (i. e., C++ with open types). The main results of comparing the overall run times of these programs with different parameter settings can be summarized as follows:

- C+++ is between 1.9 and 2.5 times slower than C++. Since C/C++ is usually regarded as a performance

```
typename Expr;          // Arithmetic expression.
Expr -> integer val;    // Value of a constant expression.
Expr -> character op;   // Operator of a compound expression.
Expr -> Expr body;      // Body of a unary expression.
Expr -> Expr left;      // Left and right operand
Expr -> Expr right;     // of a binary expression.

// Create constant/unary/binary expression.
Expr (integer v) { return Expr(@val, v); }
Expr (character o, Expr b) { return Expr(@op, o)(@body, b); }
Expr (Expr l, character o, Expr r) { return Expr(@left, l)(@op, o)(@right, r); }

// Evaluate constant/unary/binary expression.
virtual integer eval (Expr x) if (x@val) { return x@val; }
virtual integer eval (Expr x) if (x@body) {
  if (x@op == '+') return eval(x@body);
  if (x@op == '-') return -eval(x@body);
}
virtual integer eval (Expr x) if (x@left) {
  if (x@op == '+') return eval(x@left) + eval(x@right);
  if (x@op == '-') return eval(x@left) - eval(x@right);
  if (x@op == '*') return eval(x@left) * eval(x@right);
  if (x@op == '/') return eval(x@left) / eval(x@right);
}
```

Figure 1: Open type `Expr` with five attributes, three constructors, and three branches of global virtual function `eval`

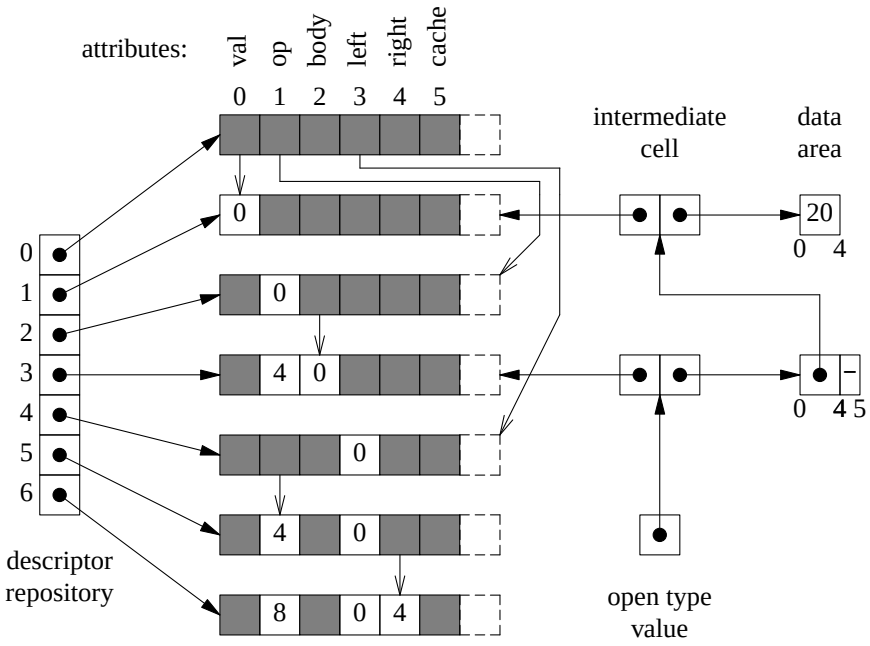

Figure 2: Attribute management

yardstick, a slowdown of this magnitude appears acceptable when considering the significantly improved flexibility provided by open types.

- C+++ is up to 2.5 times faster than Java running on the Hotspot Virtual Machine. Since Java's performance is accepted for many practical applications these days, this is a quite satisfying and encouraging result.

References

[1] C. Heinlein: "Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance." In: M. Hanus, F. Huch (eds.): *Programmiersprachen und Rechenkonzepte* (22. Workshop der GI-Fachgruppe 2.1.4; Bad Honnef, Mai 2005). Bericht Nr. 0513, Institut für Informatik, Christian-Albrechts-Universität zu Kiel, October 2005, 30–39. www.informatik.uni-kiel.de/uploads/tx_publication/2005_tr0513_01.pdf