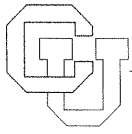
A General Property Storage Module

William M. Waite
Basim M. Kadhim

CU-CS-786-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

A General Property Storage Module

William M. Waite
Basim M. Kadhim

CU-CS-786-95 September 1995

University of Colorado at Boulder

Technical Report CU-CS-786-95
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

A General Property Storage Module

William M. Waite
Basim M. Kadhim

September 1995

Abstract

This paper describes a C module that provides unique representations for an arbitrary number of *entities*, and allows an arbitrary set of *properties* of arbitrary types to be associated with each entity. Entities can be pre-defined as well as created as the program runs; property values for pre-defined entities can be established at load time. The module exports a useful set of property query and update routines, and this set is easily extended by the user.

A property storage module for a specific application can be instantiated from a simple specification describing the requirements imposed by that application. This approach eases the task of the person designing the application, and allows them to strongly separate the property storage aspects of solutions to different subproblems.

1 Introduction

Many computer applications deal with *entities* that have *properties*. For example, a program that analyzes a directed graph deals with node and edge entities. Each edge is directed from a starting node to an ending node, and those nodes might be properties of the edge entity. Both node and edge entities might also have label properties, and so on. The object-oriented programming paradigm is well-suited to such applications: Each entity is represented by an object, and each property by a field of that object.

In most object-oriented languages, the relationships between entities and properties are established by declaring *classes* that assign properties in the form of *instance variables* (or *slots*) to a class of entities. Having to specify the mapping between groups of properties and classes of entities at compile time has the advantage of forcing the user to explicitly determine what properties each class of object must possess, but it has several disadvantages as well.

In many applications, the programmer cares only about a very small subset of the properties of an object in any given context. The class declaration of an object, however, defines all of the properties of the object and therefore exposes complexity that the programmer need not be concerned with. *Inheritance* is normally used to control this complexity, allowing the user to provide a hierarchical description in which each class only mentions the properties specially relevant to it and obtains more general properties from its ancestors.

Since the properties of most collections of entities cannot be described by a strict hierarchy, the inheritance structure is normally a directed acyclic graph rather than a tree. This leads to interesting questions about exactly which ancestor a property should be inherited from, and significantly increases the complexity of the programming language definition. Discussions of various mechanisms used to deal with this problem can be found in [4] and [5].

Having to specify the mapping between properties and entities at compile time also creates a rigid representation. For example, consider a compiler for a language like Pascal that distinguishes type identifiers from variable identifiers but provides only a single name space. The compiler must deal with type entities and variable entities, and must associate each identifier with the appropriate entity. Type entities and variable entities clearly have different properties, and would be represented by different classes of objects in an object-oriented language.

Suppose that a particular identifier is declared as a type identifier. In response to the type declaration, the compiler will create an object to represent that type entity and associate it with the identifier. Now suppose that the type identifier is erroneously used in a context where a variable identifier should appear. To avoid a cascade effect in the error analysis, after reporting the error the compiler should treat the identifier as a variable when a variable is expected and as a type when a type is expected. This may mean setting properties mapped to type entities and variable entities for a single object. Most object-oriented languages do not allow the set of properties mapped to an object to change at run time in this way.

This paper presents an alternate solution that uses typeless objects. The relationship between an entity and a property is established at run time, simply by applying an access method for the property to the object representing the entity. This avoids exposing complexity, and hence avoids the need for mechanisms to control that exposed complexity. The representation is flexible, easily accommodating arbitrary collections of properties that cannot be described by a strict hierarchy.

The cost of our solution relative to the use of classes is the cost of an associative memory (in which the location of a property value must be determined at run time), relative to the cost of a fixed memory (in which the location of a property value is known at compile time). A part of this cost is that errors detectable at compile time with classes can only be detected at run time with typeless objects.

The static typing that exists in most object-oriented languages guarantees that it is only possible to access a property of an entity if that entity actually has that property. It is not possible to make such a guarantee with typeless objects, but it *is* possible to guarantee meaningful results in any context. One approach is to use an access method that returns a boolean value indicating whether or not the property has been set for the object being queried. Another is to use an access method that takes a default value to be returned in the case that the property does not exist for the object being queried. What default value is reasonable may vary from one context to another. An object-oriented language would typically require the property value to be fixed in a way that does not depend on the context in which the value is used.

Section 2 explains how the module establishes the relationship between entities and properties and Section 3 shows how this relationship is used to access properties. Providing a specific property storage module, with the access methods described above as well as user-defined ones, involves writing a great deal of repetitive code. In Section 4, we present a specification language that allows a user to describe their data in a simple way and have the appropriate property definition module (complete with suitable access methods) generated automatically. Finally, we conclude with some comments about our experience with the approach described here and information on how to obtain the software.

2 The Underlying Mechanism

Our basic approach is to implement each entity as an associative memory. Elements of the memory are property values, and they are addressed by the corresponding property names. This is the approach taken in the LISP *association list* – a list of attribute/value pairs in which the value can be queried and modified by functions having the attribute as an argument [3].

We do not specify the implementation of the associative memory in this paper. Instead, we specify the memory interface as shown in Figure 1. The specific implementation is determined by the definitions of the `PropElt` and `Entity` structures, and the coding of the `NewKey` and `find` procedures. Some possible implementations are a linear list, a B-tree, and a hash table. Each has its advantages and disadvantages, and is appropriate given a specific pattern of accesses.

It is important to understand how the interface given in Figure 1 isolates the remainder of the program from the implementation details without wasting storage. The `PropElt` structure contains only the "overhead" information that must appear in every element of the associative memory; it does *not* specify the property value. For each property type, another structure must be defined that specifies *both* the overhead information *and* the space for the property value.

```
typedef struct PropElt {        /* Representation of a property element */
/* Linkage */
/* Which property */
} *Entry;

typedef struct Entity {         /* Representation of an entity */
/* Associative memory pointer */
} *DefTableKey;

#define NoKey (DefTableKey)0    /* Distinguished entity */

extern DefTableKey NewKey();
/* Establish a new definition
 *    On exit-
 *       NewKey=Unique definition table key
 ***/

extern int find(DefTableKey key, int p, Entry *r, size_t add);
/* Obtain a relation for a specific property of a definition
 * On entry-
 *    key=entity whose property is of interest
 *    p=selector for the desired property
 * If the definition does not have the desired property then on exit-
 *    find=false
 *    if add != 0 then r points to a new entry of size add for the property
 *    else r is undefined
 * Else on exit-
 *    find=true
 *    r points to the current entry for the property
 ***/
```

Figure 1: The Associative Memory Interface

```
typedef struct PropElt {          /* Representation of a property element */
   struct PropElt *next;             /* The next property */
   int selector;                     /* Which property */
} *Entry;

typedef struct TYPEElt {          /* Representation of a TYPE property */
  Entry next;                        /* Copy of the */
  int selector;                      /* PropElt structure */
  TYPE PropVal;                      /* Space for the value */
} *TYPEProperty;

typedef struct Entity {           /* Representation of an entity */
   Entry List;                    /* Property list pointer */
} *DefTableKey;
```

Figure 2: Defining Space

Figure 2 illustrates the definitions for an associative memory implemented as a linear list. ANSI C guarantees that if `p` is either of type `Entry` or of type `TYPEProperty`, `p->next` will be the pointer to the next structure in the list and `p->selector` will be the integer specifying which property the element contains. (This idiom of ANSI C provides an implementation of a language feature called *Type Extensions* that is a prerequisite for extensible data types in object-oriented languages [6].) The value of the property can be accessed via `p->PropVal` if `p` is of type `TYPEProperty`.

This strategy is independent of the type of value (`TYPE` can be replaced by any built-in or user-defined type identifier) and the structure uses exactly the amount of storage required by the associative memory search and the value of the property.

The procedure `find` can be implemented completely in terms of the `PropElt` structure, since this structure contains all of the information needed to implement the associative memory. Actual access to the element selected by `find` is handled by higher-level routines that know the type of the property being accessed. These routines are discussed in the next section.

3 Access Operations

Property values are accessed by routines that invoke `find` to set a pointer and then manipulate the value pointed to. For a given kind of access, there must be one routine for each

```
void
ResetTYPE(int _Property, DefTableKey key, TYPE val)
{ TYPEProperty _Item;
  { if (key == NoKey) return;
    (void)find(key, _Property, (Entry *)&_Item, sizeof(struct TYPEElt));
    _Item->PropVal = val;
  }
}
```

Figure 3: The Reset Access Mechanism

```
#define ResetPropName(key,val)      ResetTYPE(1, (key), (val))
```

Figure 4: Example Reset Access Macro

type of property to be accessed. All of these routines have the same structure, differing only in some type declarations.

Assuming the definitions of Figure 2, Figure 3 shows the structure of a routine that implements a simple access to establish the value of a property. Any built-in or user-defined type identifier representing an assignable type could be substituted for `TYPE` to obtain the routine that establishes a value of that type. The procedure is called with three parameters: the selector for the specific property whose value is to be reset (remember that there is only one routine per property *type*; any number of properties might have the same type), the key representing the entity whose property is to be reset, and the value desired.

To simplify the interfaces of the access routines, it is useful to hide the mapping from properties to selectors in a macro for each access routine/property combination. That macro simply calls the appropriate access routine with the selector chosen in the mapping. An example of such a macro is shown in Figure 4. It provides the `Reset` method for the property `PropName`, setting the property of the entity represented by `key` to a value, `val`, of type `TYPE`. This macro hides the facts that the selector `1` is used to address the `PropName` property and that the routine `ResetTYPE` actually carries out the operation.

In reviewing the implementation of the `Reset` operation shown in Figure 3, one sees that the key passed in is first tested against the value `NoKey`. The value `NoKey` is the distinguished entity exported by the interface shown in Figure 1. It represents an entity that never has any properties. As a result, applying the `Reset` operation to `NoKey` does nothing. Otherwise, `find` is used to search the associative memory of the entity given by `key` for the property identified by `_Property`. Since the fourth argument of the call is nonzero, `find` will add

```
TYPE
GetTYPE(int _Property, DefTableKey key, TYPE deflt)
{ TYPEProperty _Item;
  { if (key == NoKey) return deflt;
    if (find(key, _Property, (Entry *)&_Item, (size_t)0))
      return _Item->PropVal;
    else return deflt;
  }
}
```

Figure 5: The Get Access Mechanism

an element to the associative memory if there is none currently. In any case, upon return from `find`, `_Item` will contain a pointer to the desired property's element in the associative memory for the given entity.

Note that `_Item` is defined as a pointer to `struct TYPEElt` (see Figure 2), and when passed to `find` it is cast to hold a pointer to `struct PropElt`. Thus `find` need know only about linkage information, as discussed in the previous section. If no element currently exists, however, a `struct TYPEElt`-sized space will be reserved because of the value of `find`'s fourth argument. Linkage information will be established in this space by `find` according to the definition of a `struct PropElt`.

A more complex access function is useful to extract property information. The problem here is that a user may query the value of a non-existent property. We need to ensure that the query will return a valid value regardless of the state of the associative memory being queried. Our experience has shown that the appropriate "default" value often depends on the context of a particular query, so we require the person writing the query to supply the default value as an argument of the query itself. Again, a distinct routine is necessary for each type of property to be queried, and all of these routines have the structure shown in Figure 5.

Note that the fourth argument of `find` is `0` in Figure 5. This means that nothing will be added to the associative memory if the desired property is not found. Also, the truth value returned by `find` is used to decide whether to return the value from the element pointed to by `_Item` or the default value supplied by the user. (The default value is always returned for the distinguished entity `NoKey`, since `NoKey` never has any property values.)

4 The Property Definition Language

In the last two sections, we have shown an implementation for a general property definition module. It allows for an arbitrary number of entities, each having an arbitrary number of properties of arbitrary types. We have illustrated two simple access functions, and it should be obvious that arbitrarily complex access functions could be written. This material constitutes a *reusable design* that allows a programmer to construct a property storage module for any specific application in a straightforward way.

Although it is robust and re-usable, there is a great deal of code that must be provided for any instantiation of this design. Figure 1 describes code that need be written only once per associative memory strategy, but the code of Figures 2, 3 and 5 must be written once per property type and the code of Figure 4 must appear once for each property. (Over time one could, of course, accumulate a library of such code.)

A surprisingly small amount of information is needed to define a specific property storage module:

1. A set of named properties
2. A type for each property
3. A set of access mechanisms
4. An associative memory implementation

Once this information is available, the property storage module it defines can be created mechanically. Tremendous leverage can therefore be obtained by using a special-purpose language to express the necessary information, and having the compiler for that language produce the desired property storage module.

While the named properties and their types are going to be specific to each application, the bulk of the applications will use a small set of access mechanisms. Some of those (like `Reset` and `Get`) are so common that they should simply be provided automatically for every property. The remainder should be available as a library, and associated with particular properties by an additional specification. Finally, language facilities should be available for creating application-specific access mechanisms.

The associative memory implementation can be provided by a library module that the user can select at link time, since a very simple mechanism suffices in most cases. Thus no language facilities for associative memory specification are needed.

Figure 6 shows an example property storage module specification. It is written in *PDL*

```
Counter: int [Inc];
Storage: StorageRequired; "storage.h"
Type: DefTableKey;

int Inc(DefTableKey key)
{ if (key == NoKey) return 0;
  if (ACCESS) ++VALUE;
  else VALUE = 1;
  return VALUE;
}
```

Figure 6: Specifying a Property Storage Module

(the *P*roperty *D*efinition *L*anguage), and defines three properties. `Counter` is an integer valued property that has, in addition to the standard `Get` and `Reset` access methods, an access method called `Inc`. `Inc` is an application-specific method, whose definition appears later in the specification. The specification also defines the `Storage` property that is of type `StorageRequired`. The definition of `StorageRequired` can be found in a header file called `storage.h`, and that interface is made available to the property definition module by placing the name of the header file in double quotes in the specification. The last property defined by the specification in Figure 6 is called `Type` and is of type `DefTableKey`. No header file need be included for the definition of `DefTableKey`, since that is a type that must always be included in the generated property storage module.

The access method `Inc` is defined in C notation in the PDL specification. It is designed to take a key representing an entity, establish the value `1` if the property has not been set, and increment the value otherwise (as one would do with a counter).

`ACCESS`, `PRESENT`, and `VALUE` are macros that can be used in the body of an access method definition. These macros are defined by the PDL language, and provide a simple interface that hides the details of using `find` and addressing the property value. Their semantics are described in the PDL Manual [2], and their implementations can be seen by comparing the built-in PDL definitions of the access functions `Reset` and `Get` in Figure 7 with Figures 3 and 5.

The PDL compiler instantiates each access mechanism as a set of routines, one for each of the distinct property types to which it is applied, as discussed in Section 3. Thus there will be a single routine named `Getint` and one named `Incint` regardless of the number of properties of type `int`. The generated C functions must also have an integer argument added to the beginning of their parameter list in order to select the property.

```
void Reset(DefTableKey key, TYPE val)
{ if (key == NoKey) return;
  ACCESS; VALUE = val;
}

TYPE Get(DefTableKey key, TYPE deflt)
{ if (key == NoKey) return deflt;
  if (PRESENT) return VALUE;
  else return deflt;
}
```

Figure 7: Reset and Get in PDL

In this example, `Inc` can be applied only to the type-`int` property, `Counter`, and therefore `int` appears as the result type. Definitions of access mechanisms that can be applied to properties with a variety of types use the symbol `TYPE` instead of a specific type identifier (see Figure 7).

In addition to generating the access routines, the PDL compiler creates a C macro like the one shown in Figure 4 and described in the previous section for each access mechanism/property pair. This macro is simply a call on the appropriate routine, with the property's selector (determined by the PDL compiler) as the first argument. For example (compare with Figure 4):

```
#define IncCounter(key) Incint(1,(key))
```

The net result of the specification given in Figure 6 is a set of seven operations: one `Get` operation and one `Reset` operation for each of the three properties, plus the `IncCounter` operation. To make these operations accessible in any C program, the user need only include the header file `pdl_gen.h` generated by the PDL compiler. The PDL compiler also generates a file `pdl_gen.c`, which must be compiled and linked with the application and the module implementing `find`.

Sometimes it is useful to establish some entities at compile time, with particular property values. Figure 8 shows the notation PDL provides. Here the text in braces must be a valid C initializer for the type of the named property. The identifiers `Zero` and `MaxInt` represent entities; they are values of type `DefTableKey` that can be used as arguments to access methods to update and query their property values.

```
Zero -> Value={0};
MaxInt -> Value={32767};
```

Figure 8: Initializing Properties

5 Conclusion

We have presented a strategy for creating a general property definition module capable of representing an arbitrary number of entities, each with an arbitrary number of properties of arbitrary types. Entities and properties can exist initially or can be created during execution.

In contrast to many object-oriented programming languages, the properties of an entity are not predetermined at compile time. This simplifies the specification, and eases the handling of erroneous input, but makes compile-time consistency checks impossible.

The system described here has been used for several years to provide the definition table component for translators generated by the Eli system [1]. It had its origins in the requirement of attribute grammars that a query function always return a valid value. This led the first author to the concept of providing a default response at the point of call, since the criteria for validity often depend on the context in which the query is being made. PDL was developed as a class project by the second author, replacing a more conventional data definition language in which properties were bound to entities at compile time. Experience with constructing error recovery specifications convinced us that compile-time binding was a mistake in the context of a compiler.

The implementation of the previous data definition language had also realized each access operation with a distinct routine for each property, rather than a distinct routine for each property type as described in Section 3. We saw that that approach led to large amounts of duplicated code, which led us to the macro/routine split we use now.

In the context of translators generated by Eli, the number of properties associated with any given entity is small. We therefore use a simple linear list as an associative memory (as shown in Figure 2), and have not seen any performance penalty. We have also used this implementation of `find` in property definition modules for other applications with similar results. Although we understand how to use more complex data structures to implement the associative memory, we have thus far had no need to change our current strategy.

We believe that this approach provides a viable alternative to the use of an object-oriented language in certain circumstances. Also, since the code generated by the PDL compiler is acceptable to C++ compilers, it allows a user to gain flexibility and reduce complexity in

certain areas even when using an object-oriented language to deal with some aspects of a problem.

Both the associative memory module and the PDL compiler are available via anonymous ftp as part of the Eli system. We are currently establishing procedures for obtaining this software independent of the remainder of Eli. This task will be completed well before the paper is published, and we will include instructions on how to access it here.

Acknowledgments

This work was partially supported by the US Army Research Office under grant DAAL03-92-G-0158.

6 References

1. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. & Waite, W. M., "Eli: A Complete, Flexible Compiler Construction System," *Communications of the ACM* 35 (February 1992), 121–131.
2. Kadhim, B. M., "Property Definition Language Manual," Department of Computer Science, University of Colorado, CU-CS-95-776, Boulder, CO, July 1995.
3. McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1," *Communications of the ACM* 3 (April 1960), 184–195.
4. Schmidt, II. W. & Omohundro, S. M., "CLOS, Eiffel, and Sather: A Comparison," International Computer Science Institute, TR-91-047, Berkeley, CA, September 1991.
5. Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages," in *OOPSLA '86 Proceedings*, September 1986, 38–45.
6. Wirth, N., "Type Extensions," *ACM Transactions on Programming Languages and Systems* 10 (April 1988), 204–214.