On Object Layout for Multiple Inheritance *

William Pugh
Department of Computer Science
University of Maryland, College Park
pugh@cs.umd.edu

Grant E. Weddell
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1
gweddell@watdragon.uwaterloo.edu

*This research was supported in part by NSF grant CCR-8908900, the Natural Sciences and Engineering Research Council of Canada, by Bell-Northern Research Ltd., and by the Information Technology Research Center. The Flavors data was extracted from equipment supported by NSF research equipment grant CDA-8811952.

Abstract

We consider the problem of encoding objects for object-oriented programming languages that allow subtyping and multiple inheritance among class definitions. This is an important problem since a choice of encoding will determine the implementation for a number of common operations: extracting a property value from an object, comparing two object references for equality, and expression retyping.

We expand on earlier work in [9] in which we proposed a new algorithm for obtaining an object encoding that assigns a fixed offset to each property. This allows property values to be extracted with the same efficiency as in systems that do not provide multiple inheritance. We present both analytic and experimental evidence that suggests that this is an important performance issue and that our method works well in practice.

Index Terms: object-oriented programming languages, object encoding, compilation.

1. Introduction

Recently developed type systems for polymorphic programming languages allow a user to organize a description of objects in terms of a generalization taxonomy of class types. In this paper, we expand on earlier work in [9] that proposed a new algorithm for obtaining an object layout in cases where multiple inheritance occurs in the taxonomy. We propose a new algorithm that generalizes earlier approaches based on the idea of conversion to single inheritance [11], of table lookup [2] and a two-directional algorithm in [9].

The choice of object encoding is important since it determines:

- how property values are extracted from objects,
- how object references are compared for equality, and
- how to retype expressions at runtime.

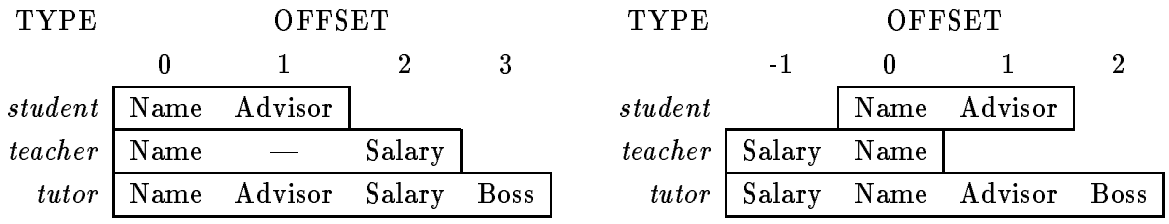
Determining a choice of behavior for an object reduces to the problem of extracting property values from an associated class object, so procedures for obtaining an object encoding may also be used to implement method dispatch.

We begin in the next section with a simple example to help motivate the results of this paper. Section 3 that follows is a survey of previous work on the problem of finding an object encoding. This includes a review of the procedures used to implement multiple inheritance in Smalltalk and in C++. We base our review on a simple type definition language introduced at the start of the section in which both class and record types may be defined. Instances of the former constitute a logical or conceptual view of a collection of objects while instances of the latter specify how objects are actually encoded in memory.

Our main results follow in Section 4 in which we present a new algorithm for obtaining an object encoding. The algorithm incorporates a generalized version of Algorithm B in [9], and can produce an object encoding in which each property is assigned a fixed offset. We demonstrate that the main advantage of this is that property value access can proceed with the same efficiency as in systems that do not provide multiple inheritance. Our analytic and experimental results further confirm that the method works well in practice, and that the efficiency of the method itself will not be a problem for cases involving class types likely to occur in practice.

In our summary comments in Section 5, we discuss a number of remaining issues and suggest some future directions of research. Part of discussion focuses on the issue of interactive use and separate compilation.

We stated above that the object encoding problem is an important performance issue. In support of this claim, we refer the reader to [12] which outlines an experiment performed on a C compiler. The results of the experiment indicate that the means of property value extraction alone can significantly affect overall performance of the compiler. This suggests that the object layout problem is an important performance issue for software systems with components that require the extensive manipulation of memory-resident data. Note that this is *not* the case if the data resides



(a) One-directional fixed offsets.

(b) Two-directional fixed offsets.

Figure 1: OBJECT LAYOUTS FOR THE UNIVERSITY TYPES.

mainly on mass store; that is, when the cost of finding a property value given an "object identifier" must include the time to read a block in which the object is encoded from mass store.[1]

2. A Motivating Example

Consider a simple university application with object types *student*, *teacher* and *tutor* with respective informal specifications given as follows.

- A student has a *name* and an *advisor* who is a teacher.
- A teacher has a *name* and a *salary*.
- A tutor is both a student and a teacher, and therefore has a *name*, an *advisor* and a *salary*. In addition, a tutor has a *boss* who is a teacher.

Suppose also that the university application has a function $Rich(T)$ that returns true if T's salary is more than \$50,000. Clearly, it should be possible to call this function with either a teacher or a tutor object as an argument (even if the result in the latter case is almost certainly false).

What code should a compiler produce to extract the salary of T? This code must work in cases where T refers to either a teacher or tutor object, and therefore will depend on the representation chosen for teacher and tutor types. One way to represent objects is simply by a list of their properties (e.g., LISP property lists). This allows us to extract an attribute of an object by scanning down the list until we find the property we are looking for. Unfortunately, this approach is clearly very expensive, even if objects have only a few properties.

In languages without multiple inheritance, we can just assign a fixed offset to each property. With multiple inheritance, we can still assign fixed offsets, although doing so may cause us to waste space, as in Figure 1(a). (Teacher objects waste the space at offset 1.)

However, if we allow objects to be placed at negative offsets, then we can layout this type structure without any wasted space, as shown in Figure 1(b). Placing properties at both positive

[1] It has been previously established for this case that the time implied by almost any choice of search strategy within this representation will be insignificant by comparison [8].

and negative offsets is termed a "two-directional record layout" in [9]. Using a two-directional record layout does not always allow us to avoid wasted space. However, in practice, we can substantially reduce the amount of wasted space over a one-directional layout, as we show in this paper. Finding a layout that wastes the minimal amount of space, for either a one or two directional layout, is an NP-hard problem. However, we present a heuristic algorithm that we have found to work well in practice.

3. Definitions and Review

Almost all existing approaches to encoding objects in the presence of multiple inheritance either require a more complicated field extraction process or require performing runtime manipulation of pointer values to perform retyping directives (called *coercions*). To the best of our knowledge, the only published exception is a two-directional scheme outlined in a preliminary version of this paper [9]. In this section, we review three of the earlier methods in detail. Borning and Ingalls proposed the first method as a way of implementing multiple inheritance for Smalltalk-80 [2], and is representative of techniques involving a table lookup when extracting field values. Bjarne Stroustrup [10] proposed the second method, which is used by most C++ compilers. This method is an extension of an earlier approach by Stein Krogdahl [7] that allows fields to be loaded from fixed offsets. The extension overcame a problem with the earlier approach which worked only for class structures in which no class could inherit a field by more than one path to a superclass. Both approaches may impose runtime overhead for coercions. One of the authors proposed the third method in his thesis [11], and was the first based on the idea of introducing unused store in records in order to align their fields.

To focus on the essential ideas, we define a simple type definition language in which both class and record types may be defined. The former are a straightforward elaboration of *type extensions* proposed in [13].

Definition 1: A *type schema* **S** consists of a set of class or record definitions with the respective forms

$$C = [\text{ VIRTUAL }] \text{ CLASS } (C_1, \ldots, C_m)\ P_1 : type_1; \ldots P_n : type_n; \text{ END} \tag{3.1}$$

and

$$R = \text{RECORD } F_1 : type_1; \ldots F_n : type_n; \text{ END.} \tag{3.2}$$

The possible types for property P or field F are given by the following grammar.

$$type ::= \text{ POINTER TO } C \mid \text{POINTER TO } R \mid R \mid \text{INT} \mid \text{STRING}(n)$$

We assume fields of type INT or POINTER will be allocated a single unit of store. To avoid name overloading and more complex type inference issues which are beyond the scope of this paper, we also assume any given property name P occurs at most once in a given type schema, and denote the associated type as $Type(P)$. (Any given field name, however, may occur in any number of different record type definitions.) A type schema consisting solely of class definitions (resp. record definitions) is called a *class schema* (resp. *record schema*). □

Our intention is that the definition of class C in (3.1) serves only to associate a set of properties with C objects, and therefore constitutes a logical or conceptual view of these objects. Following standard practice, we view C objects to also be C_i objects, and to have a value for property P only when $P = P_i$, for some $1 \leq i \leq n$, or when C_i objects have a value for property P. (In the latter case, P is usually called an *inherited* property.) Also following standard practice, we assume that all objects are created with respect to a single non-virtual class—that the set of non-virtual classes in a type schema is an exhaustive enumeration of the kinds of objects that can exist at runtime. (A class is non-virtual if the keyword "VIRTUAL" is not mentioned is its definition.)

In contrast, the definition of record R in (3.2) is intended to suggest the actual layout in memory of a sequence of fields. We use record types in this paper to indicate how objects and their property values are actually encoded in memory. The presumed relationship to class types is straightforward: a field name which is the same as a property name encodes values for the property, while a record name which is the same as a class name specifies the internal encoding of objects created from the class.

A formal specification of the university object types of the previous section is given by the class schema in Table 1 (we refer to this as the "university schema" in the remainder of the paper). Observe that an additional virtual class Person is introduced as an immediate superclass of Student and Teacher in order to factor the common Name property. Note that, since class Person is virtual, a person object must also be (at least) a student or teacher object. The student and teacher classes are themselves declared as immediate superclasses of the class Tutor. Consequently, each tutor object will have the required four property values: a Boss since the definition of the tutor class mentions this property, an Advisor and Salary since the tutor class inherits these properties from its immediate superclasses, and a (single) Name since class Tutor inherits this property from class Person indirectly.

Table 1: THE UNIVERSITY CLASS SCHEMA.

Person = VIRTUAL CLASS () Name: STRING(20); END
Student = CLASS (Person) Advisor: POINTER TO Teacher; END
Teacher = CLASS (Person) Salary: INT; END
Tutor = CLASS (Student, Teacher) Boss: POINTER TO Teacher; END

Recall our earlier comments that a choice of object encoding determines how property values are extracted from an object, how object references are compared for equality, and how coercion operations are implemented at runtime. A simple expression language comprised of these operations

which will suffice for our illustrative purposes is given as follows.

$$\begin{array}{lcll} exp & ::= & c & \text{(denotes a } C \text{ object)} \\ & | & exp \rightarrow P & \text{(property value access)} \\ & | & exp = exp & \text{(object equality)} \\ & | & exp \text{ AS } C & \text{(type coercion)} \end{array}$$

For an example relating to the university schema, assume that a student object *student* has a tutor object as the value of its advisor property. The expression

$$((student \rightarrow \text{Advisor}) \text{ AS Student}) \rightarrow \text{Advisor} \rightarrow \text{Name} \tag{3.3}$$

evaluates to the "name of the advisor of the advisor of *student*" in the following manner. First, the advisor property value for the initial student object is retrieved. Since type analysis of the university schema can only determine that the result is a teacher object, a coercion operation to the Student class is necessary before the advisor property value for the tutor object is retrieved.[2] Finally, a character string that is the name of the advisor of this tutor object is returned. For another example, the expression

$$(student \rightarrow \text{Advisor}) = (tutor \rightarrow \text{Advisor}) \tag{3.4}$$

determines if a given student object *student* and tutor object *tutor* have the same advisor.

3.1 Multiple Inheritance in Smalltalk-80

In Smalltalk-80, an internal identifier is maintained for each object that is essentially the address of a record in memory encoding the object's property values (properties are called *instance variables* in Smalltalk-80). A field is added at the start of each record to record the internal identifier, also an address, of each object's class object. A class object stores the compiled functions, called *methods*, peculiar to objects of that class together with the address of at most one immediate superclass. Any compiled method accessing the value of a property assumes that the value can always be found at a particular offset within the record for any object for which the property is defined. This is easy to ensure when multiple inheritance is not permitted by always locating the fields for properties defined on a class prior to those for properties defined on its subclasses.

The mechanism proposed in [2] to support multiple inheritance uses this same scheme whenever the field offsets of any properties accessed by a method are not ambiguous. If this is not the case for some method (if the object layout for two different classes locates fields for some property accessed by the method at different offsets), then a separate copy of the method is compiled and stored with each class object for which the method is defined.

The location of a property within a record type depends on the order in which immediate superclasses are mentioned in the definition of a class. Properties inherited from the *first* immediate superclass are followed by those of the class itself, and then by those inherited from the remaining immediate superclasses not already present. According to this scheme, record types specifying the layout of objects for the university schema would appear as in Table 2.[3] Observe in the case of

[2] As in [13], we would like to support coercion operations that cannot be guaranteed safe at compile time.

[3] This is not quite true in Smalltalk-80 for two reasons: instance variables are not typed, and strings must usually

Table 2: Object layout in Smalltalk-80.

```
Person = RECORD
    Tag: POINTER TO MethodTable;
    Name: STRING(20);
END

Student = RECORD
    Tag: POINTER TO MethodTable;
    Name: STRING(20);
    Advisor: POINTER TO Teacher;
END

Teacher = RECORD
    Tag: POINTER TO MethodTable;
    Name: STRING(20);
    Salary: INT;
END

Tutor = RECORD
    Tag: POINTER TO MethodTable;
    Name: STRING(20);
    Advisor: POINTER TO Teacher;
    Boss: POINTER TO Teacher;
    Salary: INT;
END
```

the Tutor record that the Advisor field occurs prior to the Salary field since the Student class is the first immediate superclass mentioned in the definition of the Tutor class. As a consequence, any method defined for the Teacher class that accesses Salary values must have separate compiled versions attached to the Teacher and Tutor class objects.

Another way of understanding this is to consider that there are two ways of compiling operations accessing property values. The first is an optimal indexed load that may be employed whenever the offset of the field encoding the property is unambiguous. A simple indexed load, for example, can be used for all "$\rightarrow$" operators in expressions (3.3) and (3.4) above, or in the expression

$$tutor \rightarrow \text{Salary}.$$

The second is a more costly table lookup that uses a "tag" field value at the start of a record that identifies the record type, together with an indication of the property itself, to find the actual field location for the property in some suitably organized table. This would be necessary for the "$\rightarrow$" operator in the expression

$$teacher \rightarrow \text{Salary}$$

since, at runtime, *teacher* might refer to either a Teacher object or Tutor object.

With this approach to object layout, object identifiers can correspond uniformly to the address of the *start* of the record encoding the object. Consequently, both "=" and "AS" operators have

be encoded as separate objects with their own "arrayed" instance variable.

optimal overhead at runtime: comparing two pointer values in the first case, and doing nothing at all in the second.

Note that the Smalltalk arbitration procedure for choosing the ordering of fields within records is based on the order in which immediate superclasses occur in class types. Since this determines which of the "$\rightarrow$" operators will require a more expensive table lookup at runtime, a preferable means of arbitration might be based on the relative frequency of execution of these operators.

3.2 Multiple Inheritance in C++

In C++, the internal identitier of an object is also the address of a record encoding the object's property values (properties in this case are called *members*). However, unlike the Smalltalk scheme, the mechanism proposed in [10] to support multiple inheritance allows an optimal indexed load to be used for *all* operations accessing property values. This is accomplished at the cost of additional overhead for some coercion operations at runtime. In particular, it might be necessary to add a (possibly negative) constant offset to a pointer value, or to obtain another pointer value to a different location within the same record by accessing the contents of a special field within the record.

The kind of overhead that will be necessary for a given "AS" operator depends indirectly on the declaration of *virtual superclasses* by the user. In particular, a superclass C mentioned in the definition of class C' should be declared as a virtual superclass of C' by the user if (1) C' has a subclass C'' for which a subclass path exists from C that does *not* pass through C', and (2) if a single copy of C's properties are to be inherited by C''. For example, since there are two subclass paths to Tutor from Person (one via Student and the other via Teacher), the class Person in Table 1 should be declared as a virtual superclass in the definitions of classes Student and Teacher to ensure that objects in the Tutor class have a single Name property value.

Record types specifying an object layout for the non-virtual classes in the university schema that correspond to the layout determined by the procedures in [10] are given in Table 3. Observe that an additional "CTemplate" record type exists for each class C. Like the Smalltalk-80 approach, the format of CTemplate depends on the order in which immediate superclasses are mentioned in the definition of C. In particular, a field named "C_iPart" for each immediate superclass C_i of C is allocated in this order, followed by fields encoding the uninherited properties of C. The type of field "C_iPart" is either "C_iTemplate", if C_i is *not* declared as a virtual superclass of C, or "POINTER TO C_iTemplate" otherwise. The latter circumstance creates what we referred to as a special field above. Record type C for a non-virtual class C is then assigned a "CTemplate" field followed by "C'Template" fields for each superclass C' of C for which there exists another (not necessarily distinct) superclass C'' of C declaring C' as a virtual superclass. For example, the Tutor record type has a PersonStore field following a TutorStore field since class Student, for example, is a superclass of Tutor declaring Person as a virtual superclass.

This approach to object layout requires each "POINTER TO C_jTemplate" field occurring in the record type for a new C record to be initialized with the address of the "C_jStore" field in the C record. To help clarify this, Figure 2 illustrates an instance of each C record type for the university class schema. The property values indicate, for example, that Mary is the advisor of both Jane and

Table 3: Object layout in C++.

```
Student = RECORD
    StudentStore: StudentTemplate;
    PersonStore: PersonTemplate;
END

Teacher = RECORD
    TeacherStore: TeacherTemplate;
    PersonStore: PersonTemplate;
END

Tutor = RECORD
    TutorStore: TutorTemplate;
    PersonStore: PersonTemplate;
END
```

```
StudentTemplate = RECORD
    PersonPart: POINTER TO PersonTemplate;
    Advisor: POINTER TO Teacher;
END

TeacherTemplate = RECORD
    PersonPart: POINTER TO PersonTemplate;
    Salary: INT;
END

TutorTemplate = RECORD
    StudentPart: StudentTemplate;
    TeacherPart: TeacherTemplate;
    Boss: POINTER TO Teacher;
END

PersonTemplate = RECORD
    Name: STRING(20);
END
```

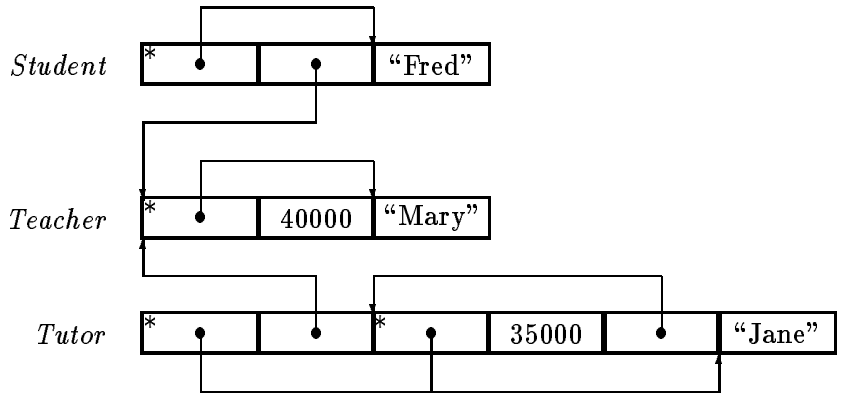


Figure 2: EXAMPLE OBJECT LAYOUT IN C++.

Fred, and that Jane (who is a tutor) is her own boss. The special fields needed for this case are marked with an asterisk.

Assume compile time type analysis can determine that expression *exp* must evaluate to a C object. If class C inherits property P from a proper superclass C', then the C++ approach to object encoding requires that a compiler first transforms any expression of the form "$exp \rightarrow P$" by introducing additional coercion operators to C'. This is also true for expressions of the form "$exp_1 = exp_2$" if the types of the argument expressions differ. For example, the expressions

$$tutor \rightarrow \text{Salary}$$
$$(tutor \rightarrow \text{Advisor}) = student$$

would require the addition of four coercion operators as follows:

$$(tutor\ \text{AS}_1\ \text{Teacher}) \rightarrow \text{Salary}$$
$$(((tutor\ \text{AS}_2\ \text{Student}) \rightarrow \text{Advisor})\ \text{AS}_3\ \text{Person}) = (student\ \text{AS}_4\ \text{Person}).$$

Operators AS_1 and AS_2 are added in order to access the Salary and Advisor property values of the Tutor object. Operators AS_3 and AS_4 are added to convert the arguments of the equality operator to a common type.

The compiler must then replace each AS operator by either no code at all, code that adds a (possibly negative) constant offset to a pointer value, or code that loads the value of a special field. An example of the first case is AS_2 above, since the expression *tutor* will evaluate to something that already "looks" like a Student object. An example of the second case is AS_1 in which a constant equal to the size of a StudentTemplate record must be added to (the result of evaluating) *tutor* in order to produce an address of something that looks like a Teacher object. Finally, operators AS_3 and AS_4 above are examples of the last case in which code to load the value of a PersonPart field is generated.

In addition to its complexity, a main limitation of this approach to object layout is that it does not directly support a casting operation through virtual superclass links. For example, if a Person object *person* is also (at least) a Student object, and we wish the value of the expression

$$(person \text{ AS Student}) \rightarrow \text{Advisor},$$

then we must relocate the start of the Student record by some other means since we cannot determine in general where a PersonTemplate field occurs in a record.

3.3 Object Layout by Conversion to Single Inheritance

The C++ approach to object layout may require allocating additional store for special fields to enable navigating within an object encoding. This is necessary to ensure that fields can be loaded from a fixed offset. Another approach explored by one of the authors in his thesis [11] also allocates additional store to align fields. In this case, however, navigating within an object is never necessary.

This earlier work considered how to convert a class hierarchy with multiple inheritance into a class hierarchy with single inheritance in a manner that would result in a minimal number of superfluous inherited property values. The conversion procedure essentially involved a depth-first-search of the class hierarchy, starting from the most general classes. Based on object count and property size estimates, disjoint superclass paths are merged to produce an alternative hierarchy with single inheritance. This is done in such a way as to preserve any superclass relationships in the original hierarchy. For example, if these estimates imply that there are more Student objects than Teacher objects, then Table 4 lists the new class types that would be output by this procedure when applied to the university schema. Observe that the important change made to the original schema in Table 1 was to replace the immediate superclass Person in the definition of Teacher by class Student. The net effect of this will be that new Teacher objects will be allocated space for an unnecessary Advisor property value.

Table 4: Object layout by conversion to single inheritance.

Person = VIRTUAL CLASS () Name: STRING(20); END
Student = CLASS (Person) Advisor: POINTER TO Teacher; END
Teacher = CLASS (Student) Salary: INT; END
Tutor = CLASS (Teacher) Boss: POINTER TO Teacher; END

Like the Smalltalk-80 approach to object layout, object identifiers can correspond uniformly to the address of the start of the record encoding the object, and therefore "=" and "AS" operators

have the same optimal overhead at runtime. However, by admitting the possibility of unused store, this approach allows all property value access to be implemented as an optimal indexed load. The main disadvantage with this approach is that it requires additional statistical information beyond the user defined class types as input in order to be effective. Also, experiments have revealed that the percentage of unused store in records can become a significant overhead when multiple inheritance is used more extensively in a type schema.

4. Multidirectional Record Layout

In this section, we present a new algorithm for obtaining an object encoding that generalizes the above approaches based on the ideas of conversion to single inheritance and table lookup, as well as a two-directional approach considered in [9]. The algorithm involves applying a sequence of two procedures. The first of these procedures, a generalized version of Algorithm B in [9], is responsible for computing an *object layout* for a given type schema **S**. The layout consists of *direction* and *offset* assignments for each of the properties occurring in **S**, and is derived by carefully *placing* properties in a way that ensures minimal overhead of wasted store in records encoding objects. The second procedure then generates a set of record types according to the original class definitions in **S** and the object layout computed by the first procedure. The new record types constitute a specification of the object encoding for the non-virtual classes.

The main reason we give the second procedure is to clarify how our approach can be easily adapted for use in a preprocessor. For this reason, we do not attempt to prove any results about the procedure in this paper. Our analysis and our experiments focus instead on evaluating the performance of the first procedure.

In addition to a type schema, this first procedure is also supplied with a positive integer n that specifies a bound on the number of directions permitted for object layout. The object layout produced by this procedure for a particular value of n relates to the layout produced by methods reviewed earlier as follows.

- If $n = 1$, then fields encoding the properties occur at a fixed positive offset. This essentially manifests the conversion to single inheritance approach.

- IF $n = 2$, then fields encoding the properties also occur at a fixed positive offset. In this case, however, it must be possible for pointers to address memory at locations prior to the store allocated for records. This is a variation of the two-directional method, and requires that the object language supports pointer arithmetic as well as pointer types.

- If $n > 2$, then a table lookup will be needed when extracting values for fields encoding properties placed in a direction greater than 2 by the first procedure (assuming the object language supports pointer arithmetic). Fields encoding properties placed in the first two directions will still occur as a fixed positive offset. This corresponds to a more sophisticated Smalltalk-80 approach in which we allow additional overhead for (some of the) "$\rightarrow$" operators in order to avoid any need for wasted store to align fields.

A user can therefore effect tradeoffs between store costs and execution efficiency by supplying different values of n. However, our experiments suggest that this is not an issue in cases where the object language supports pointer arithmetic, since a bound of 2 appears to be sufficient to ensure an acceptable overhead of wasted store for type schema likely to occur in practice. (For this reason, our second procedure given later in this section for generating record definitions does not consider the case above where $n > 2$.)

To illustrate our approach, the first (resp. second) column of Table 5 lists record types specifying the object encoding for the university schema generated by the second procedure when given an object layout produced by the first procedure for $n = 1$ (resp. $n = 2$). In the first column, the field SkipOff2 is needed to ensure Advisor fields occur at a fixed offset. In the second column, the field ShiftOff1 is needed to ensure both Name and Salary fields occur at a fixed offset (recall that we assume INT and POINTER fields are allocated the same amount of store). However, unlike the previous case, it is not necessary to allocate store for this field since it occurs at the start of the record definition. Thus, a pointer to a new Student record will address memory at a position prior to the store allocated for the record. (This is why pointer arithmetic is needed when $n = 2$.) To help clarify this latter case, Figure 3 illustrates corresponding instances of the records in Figure 2.

In general, assume an expression of the form "ALLOCATE(m)" returns a pointer to newly allocated store of size m units, and that *Shift*(R) denotes a count of the number of "ShiftOff$i$" fields occurring in the definition of record R. An expression allocating store for an R record can have the form[4]

$$\text{ALLOCATE(}\ \text{SIZEOF}(R) - \mathit{Shift}(R)\ \text{)} - \mathit{Shift}(R).$$

If the R record is intended to encode an R object, then the returned address will usually qualify as an object identifier for the R object. In this way, "AS" operators will involve no overhead at run-time and "=" operators simply require comparing two pointer values for equality.

An exception to this happens in cases where two different records have no fields in common. For example, if the definition of the Student class did not mention class Person as a superclass, then the student record definitions in Table 5 would not have Name fields. Consequently, it becomes possible for the Fred and Mary records in Figure 3 to overlay one another, and therefore that *Student* and *Teacher* can address the same location in memory (i.e. the Advisor field encoding the Advisor property value for Fred). However, this is not a problem for most real situations since the record types specifying representations for classes will usually require a common "Tag" field for use in run-time type checking, for method dispatch, and so on.

With our approach, fields encoding any pointer or pointer-sized properties are assigned a fixed offset and therefore "→" operators can be implemented as an indexed load in most cases. This is usually true for other properties as well, but there are circumstances in which more lengthy values

[4]The two-directional method proposed in [9] assumed that fields values could be extracted from a record at a negative offset. Consequently, this earlier method was more difficult to adapt for use in preprocessors for existing languages. Cormack [4] suggested the alternative used here in which a constant value is *subtracted* from the address of newly allocated object store. This shifts fields placed in the second direction to a positive offset.

Table 5: n-DIRECTIONAL OBJECT LAYOUT.

$n = 1$	$n = 2$

Student = RECORD
  Name: STRING(20);
  SkipOff2: INT;
  Advisor: POINTER TO Teacher;
END
 	
Teacher = RECORD
  Name: STRING(20);
  Salary: INT;
END
 	
Tutor = RECORD
  Name: STRING(20);
  Salary: INT;
  Advisor: POINTER TO Teacher;
  Boss: POINTER TO Teacher;
END
	Student = RECORD
	  Advisor: POINTER TO Teacher;
	  Name: STRING(20);
	END
	
	Teacher = RECORD
	  ShiftOff1: INT;
	  Name: STRING(20);
	  Salary: INT;
	END
	
	Tutor = RECORD
	  Advisor: POINTER TO Teacher;
	  Name: STRING(20);
	  Salary: INT;
	  Boss: POINTER TO Teacher;
	END

for some properties require a level of indirection in their encoding within a record. In these cases, "$\rightarrow$" operations will require two indexed loads in order to access the value of a property (we discuss this point more fully at the end of this section when we present the second procedure).

4.1 Schema analysis and terminology.

Computing an object layout requires some initial analysis of the class schema. Part of this analysis is to determine a list of *property sets* that exhaustively enumerates the various combinations of properties for which objects may have corresponding values at runtime. In our case, each non-virtual class in the original type schema should therefore contribute one such property set. A formal definition of property sets, together with some related terminology and notation, are given in the following.

Definition 2: A property P is in the *property set* for class C, denoted $Pset(C)$, if and only if a C object *must* have a value for property P. The *property set list* for type schema **S** consists of a property set for each non-virtual class C defined in **S**. The set of all properties occurring in a given property set list is denoted *props*, where the property set list and associated type schema are

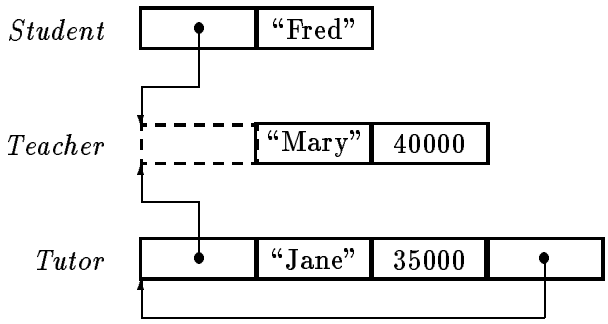


Figure 3: EXAMPLE 2-DIRECTIONAL OBJECT LAYOUT.

understood from context. By additional convention, we use x, y and z (possibly subscripted) to denote instances of *props*.

For each $x \in props$, the *record count* for x, written $Rcnt(x)$, denotes a count of the number of property sets (determined by context) in which x occurs. For each pair of distinct properties $x, y \in props$, the *join count* for x and y, written $Jcnt(x, y)$, denotes a count of the number of property sets in which both x and y occur. Note that this latter relationship is symmetric (i.e. $Jcnt(x, y) = Jcnt(y, x)$), and that all data, including the property set list, can be easily computed by scanning a given type schema. □

For example, the property set list for the university schema contains the following three property sets (one for each of the three non-virtual class definitions):

{Name, Advisor}, {Name, Salary} and {Name, Advisor, Salary, Boss}.

Also, for example, $Rcnt(\text{Name}) = 3$, and $Jcnt(\text{Name}, \text{Advisor}) = 2$.

Lemma 1: For any combination of distinct properties x, y and z in *props*, if $Jcnt(x, y) = Rcnt(y)$ and $Jcnt(x, z) = 0$, then $Jcnt(y, z) = 0$.

Proof. Assume $Jcnt(y, z) \neq 0$. Then there exists a property set S containing properties y and z. If $Jcnt(x, y) = Rcnt(y)$ then x must occur in any property set in which y occurs. Thus S must contain x and therefore $Jcnt(x, z) \neq 0$. □

Definition 3: An *n-directional object layout* for type schema **S** is a pair of total functions

$$Dir : props \rightarrow \{1, \ldots, n\} \text{ and } Off : props \rightarrow \{1, 2, \ldots\}.$$

The layout is *legal* if and only if, for any pair of distinct properties x and y in *props*, either $Dir(x) \neq Dir(y)$, $Off(x) \neq Off(y)$ or $Jcnt(x, y) = 0$. The layout is *perfect* if and only if it is legal and, for every property set $\{x_1, \ldots, x_l\}$ in the property set list for **S**, whenever there exists $1 \leq i \leq l$ such that $Off(x_i) > 1$, there exists $1 \leq j \leq l$ such that $Dir(x_j) = Dir(x_i)$ and $Off(x_j) = Off(x_i) - 1$. □

For example, *Dir* and *Off* functions defining a legal 1-directional object layout for the university schema are given as follows.

$$\begin{array}{llll} Dir(\text{Name}) = 1 & Dir(\text{Salary}) = 1 & Dir(\text{Advisor}) = 1 & Dir(\text{Boss}) = 1 \\ Off(\text{Name}) = 1 & Off(\text{Salary}) = 2 & Off(\text{Advisor}) = 3 & Off(\text{Boss}) = 4 \end{array}$$

The layout is not perfect since the property set $Pset(\text{Student}) = \{\text{Name, Advisor}\}$ has no property x_j such that $Dir(x_j) = Dir(\text{Advisor})$ and $Off(x_j) = Off(\text{Advisor}) - 1$ (which implies that records encoding teacher objects will have wasted store). A perfect 2-directional object layout for the same schema is given as follows.

$$\begin{array}{llll} Dir(\text{Advisor}) = 2 & Dir(\text{Name}) = 1 & Dir(\text{Salary}) = 1 & Dir(\text{Boss}) = 1 \\ Off(\text{Advisor}) = 1 & Off(\text{Name}) = 1 & Off(\text{Salary}) = 2 & Off(\text{Boss}) = 3 \end{array}$$

(Note that these layouts correspond to the record definitions in Table 5.)

We now define an implication relation over the properties occurring in a given type schema **S**, and a *conflict* graph in which the properties occur as vertices. This information will largely determine the order in which properties will be assigned a direction and offset by a procedure for computing an object layout.

Definition 4: The *implies relation* induced by type schema **S** consists of all ordered pairs (x, y) of distinct properties in *props* such that $Rcnt(x) = Jcnt(x, y)$ and either $Jcnt(x, y) < Rcnt(y)$ or y compares less than x in lexicographical order. We write $x \Rightarrow y$ as shorthand for the condition that (x, y) is in the implies relation induced by **S**.

The *conflict graph* $G = (props, E)$ induced by **S** includes each property x in *props* as a vertex, and has edge $\{x, y\}$ in E if and only if $Jcnt(x, y) \neq 0$ and neither $x \Rightarrow y$ nor $y \Rightarrow x$ is true. We write $x \rightleftharpoons y$ as shorthand for the condition that $\{x, y\} \in E$. □

Informally, $x \Rightarrow y$ is true if, whenever an x property value exists for an object, a y property value must also exist for the object. The slight complication in the definition of the implies relation concerning "lexicographical order" is simply to ensure that the graph imposed by the relation is acyclic (and therefore admits a topological sort). Also note that the implies relation is transitive. In the case of the conflict graph, we will see that a perfect object layout will often require properties x and y to be placed in a different direction if $x \rightleftharpoons y$ is true.

4.2 Computing an n-directional object layout.

It is straightforward to compute a perfect n-directional object layout for a given type schema **S** if an n-coloring for the conflict graph induced by **S** is already provided. The proof to the following theorem gives a procedure for doing this which is based on the idea of placing properties in a topological order of the implies relation induced by **S**. The procedure we propose for computing an object layout is based on the same general approach.

Theorem 1: If the conflict graph G induced by type schema **S** is n-colorable, then a perfect n-directional object layout exists for **S**.

Proof. Assume an n-coloring exists for the conflict graph $G = (props, E)$ induced by **S**. In particular, assume there exists a total function $Color : props \rightarrow \{1, \ldots, n\}$ such that $Color(x) \neq Color(y)$ whenever $\{x, y\} \in E$, and consider an object layout for **S** computed by the following algorithm.

> **Algorithm A.** In a topological order $(x_1, x_2, \ldots, x_m)$ of the implies relation induced by **S** (in which later properties imply earlier properties), *place* each property x_i by first updating $Dir(x_i)$ with the value $Color(x_i)$ and then assigning $Off(x_i)$ the value
>
> $$\min\{\ 1 \leq k \mid \forall 1 \leq j < i, Dir(x_j) \neq Dir(x_i) \text{ or } Off(x_j) \neq k \text{ or } Jcnt(x_j, x_i) = 0\ \}. \qquad (4.1)$$

Algorithm A clearly computes a legal n-directional object layout for **S**. We now prove that the layout must also be perfect.

Assume, conversely, that the layout is not perfect. Then there must exist a property set $\{y_1, \ldots, y_l\}$ in the property set list for **S** and integer $1 \leq i \leq l$ such that $Off(y_i) > 1$ and, for all $1 \leq j \leq l$, $Off(y_j) \neq Off(y_i) - 1$ whenever $Dir(y_j) = Dir(y_i)$. The computation of the offset assignment for y_i (4.1) implies that there exists at least one property z placed (chronologically) before y_i for which (1) $Dir(z) = Dir(y_i)$, (2) $Off(z) = Off(y_i) - 1$ and (3) $Jcnt(z, y_i) \neq 0$. A consequence of the first two of these conditions is that $z \neq y_j$ for all $1 \leq j \leq l$. Thus, $Rcnt(y_i) > Jcnt(z, y_i)$ and therefore $y_i \Rightarrow z$ is false. Since z is placed before y_i, z occurs earlier in the above-mentioned topological order, and therefore $z \Rightarrow y_i$ is also false. But then condition (3) implies $\{z, y_i\} \in E$, and therefore $Color(z) \neq Color(y_i)$, by choice of the function $Color$, and therefore $Dir(z) \neq Dir(y_i)$—a contradiction with condition (1). □

Since a 1-coloring exists for an arbitrary graph if and only if the graph contains no edges, and a 2-coloring exists if and only if the graph is bipartite, it is straightforward to devise a simple procedure based on Algorithm A that will compute perfect 1 or 2-directional object layouts in these cases. The first step of this procedure would construct the conflict graph and then n-color the graph in a way that reduced as much as possible the number of edges connecting properties assigned the same color. The properties would then be placed according to the above algorithm, leaving wasted space when necessary. Unfortunately, this procedure does not work very well for many real situations. The problem is that it does not consider the ramifications of *merging* two properties by assigning them the same direction and offset. For example, consider a type schema consisting of the following class definitions.

```
C1 = CLASS () A: INT; END
C2 = CLASS () B: INT; END
C3 = VIRTUAL CLASS () C: INT; END
C4 = CLASS (C1, C3) END
C5 = CLASS (C2, C3) END
```

A perfect 1-directional object layout for the schema is given as follows.

$$Dir(\mathrm{A}) = 1 \quad Dir(\mathrm{B}) = 1 \quad Dir(\mathrm{C}) = 1$$
$$Off(\mathrm{A}) = 1 \quad Off(\mathrm{B}) = 1 \quad Off(\mathrm{C}) = 2$$

However, since the conflict graph induced by this schema contains the two edges $\{A, C\}$ and $\{C, B\}$ (and is therefore not 1-colorable), the procedure would fail to find this layout if supplied with a value of $n = 2$.

This illustrates a case in which a perfect 1-directional object layout exists for a schema, but where the conflict graph induced by the schema is not 1-colorable. In fact, this circumstance holds for any value of n.

Theorem 2: For any positive integer n, there exists a class schema **S** with a perfect 1-directional object layout, but where the conflict graph induced by **S** is not n-colorable.

Proof. The case for $n = 1$ is already given. For $n > 1$, assume $G' = (V', E')$ denotes an $(n+1)$–clique (i.e. a graph with $n + 1$ vertices and an edge between every pair of vertices), and let **S** consist of a class definition for each vertex x in V' with the form

$$C_x = \text{VIRTUAL CLASS } () \; x\text{: INT; END}$$

and for each edge $\{x, y\}$ in E' with the form

$$C_{xy} = \text{CLASS } (C_x, C_y) \; xy_3\text{: INT; } \ldots\text{; } xy_{n+1}\text{: INT; END.}$$

Thus, for every pair of vertices x and y in V', $Jcnt(x, y) = 1$, and for every vertex x in V', $Rcnt(x) > 1$ (since there must be at least two edges incident to x in G'). This implies that G' is a subgraph of the conflict graph G induced by **S**, and therefore that G is not n-colorable.

A perfect 1-directional object layout for **S** is easily obtained as follows. First, for an arbitrary permutation $(x_1, \ldots, x_{n+1})$ of V', let $Off(x_i) = i$. Now consider the offset for properties xy_k occurring in the definition of the "edge" class C_{xy}. Since these properties occur in a single property set, they can be safely assigned any offset in the range 1 to $n + 1$ not already assigned to x or y, and since there are $n - 1$ of them, every offset in this range can be assigned a property. □

Consider attempting to compute a 1-directional object layout for the type schema consisting of classes C1 to C5 above. If properties A and B are assigned offset 1, it becomes safe to place property C at offset 2. To properly handle such situations, the conflict graph should therefore be colored only as properties are placed, and when properties are merged (i.e. assigned the same direction and offset), the effect this has on the conflict graph should be noted immediately.

We have incorporated this idea in the procedure shown in Figure 4. Essentially, the procedure keeps track of the *first* property x assigned a given direction and offset. If a later property y is also assigned the same direction and offset, the procedure updates the *Jcnt* function to achieve the effect that would be obtained by replacing all occurrences of y with x in the original property set list for the input type schema.

Theorem 3: Given type schema **S** and integer $n > 0$, procedure PLACEPROPS computes a valid n-directional object layout for **S**.

procedure PLACEPROPS(**S**, n)

(initialization)

for each $x \in props$, $ImpliesCount(x) := |\{y \in props \mid x \Rightarrow y\}|$;
$boundary := \{x \in props \mid ImpliesCount(x) = 0\}$;
$unplaced := props$;
$placed := \emptyset$;

(loop body—terminate when $boundary = \emptyset$**)**

Step 1. (*property selection*)
$frustrated := \{x \in boundary \mid \forall 1 \leq i \leq n, \exists y \in placed \text{ s.t. } Dir(y) = i \text{ and } x \rightleftharpoons y\}$;
$desperate := \{x \in boundary \mid \exists y \in placed \text{ s.t. } x \rightleftharpoons y\} - frustrated$;
if $desperate \neq \emptyset$
then find $x \in desperate$ s.t. $\forall y \in desperate$, $Rcnt(y) \leq Rcnt(x)$
else if $frustrated \neq \emptyset$
then find $x \in frustrated$ s.t. $\forall y \in frustrated$, $Rcnt(y) \leq Rcnt(x)$
else find $x \in boundary$ s.t. $\forall y \in boundary$, $Rcnt(y) \leq Rcnt(x)$;
remove x from $boundary$ and $unplaced$;
for each $y \in props$ s.t. $y \Rightarrow x$,
decrement $ImpliesCount(y)$;
if $ImpliesCount(y) = 0$ then add y to $boundary$;

Step 2. (*property placement*)
$k := \min\{\ 0 \leq i \mid \exists 1 \leq j \leq n \text{ s.t. } i = |\{y \in placed \mid Dir(y) = j \text{ and } x \rightleftharpoons y\}|\ \}$;
$Dir(x) := \min\{\ 1 \leq i \leq n \mid k = |\{y \in placed \mid Dir(y) = i \text{ and } x \rightleftharpoons y\}|\ \}$;
$Off(x) := \min\{\ 1 \leq i \mid \forall y \in placed, Dir(y) \neq Dir(x) \text{ or } Off(y) \neq i \text{ or } Jcnt(x, y) = 0)\ \}$;

Step 3. (*property merging*)
if there exists $y \in placed$ s.t. $Dir(y) = Dir(x)$ and $Off(y) = Off(x)$
then for each $z \in unplaced$, $Jcnt(z, y) := Jcnt(y, z) := Jcnt(y, z) + Jcnt(x, z)$;
else add x to $placed$;

Figure 4: A PROCEDURE FOR COMPUTING AN OBJECT LAYOUT.

Proof. The procedure uses a simple abstraction of Algorithm T in [6] to ensure each property in *props* is assigned a direction and offset according to a topological order of the implies relation induced by **S** (first placing properties that imply no others). What remains is to show that the layout is legal. Since Step 3 ensures that the set *placed* includes a property for each combination of *Dir* and *Off* values assigned to previously selected properties, this follows by a simple inspection of Step 2 and the correctness of Algorithm A. □

Lemma 2: In the loop body of procedure PLACEPROPS, if $(desperate \cup frustrated) = \emptyset$, then there is no property $x \in unplaced$ and property $y \in placed$ such that $x \rightleftharpoons y$ is true.

Proof. If $(desperate \cup frustrated) = \emptyset$, then $z \rightleftharpoons y$ does not hold for any properties $z \in boundary$ and $y \in placed$. Assume there exists properties $x' \in unplaced$ and $y' \in placed$ such that $x' \rightleftharpoons y'$ is true. Since $x' \in unplaced$, there exists $z' \in boundary$ such that $x' \Rightarrow z'$ is true. Since $x' \rightleftharpoons y'$ is true, $Jcnt(x', y') > 0$, and since $x' \Rightarrow z'$ is true, $Jcnt(y', z') > 0$ by Lemma 1. Thus, since $z' \rightleftharpoons y'$ does not hold and $y' \in placed$, $z' \Rightarrow y'$ must hold, and therefore $x' \Rightarrow y'$ holds by transitivity. But then $x' \rightleftharpoons y'$ cannot hold—a contradiction. □

Theorem 4: If the conflict graph induced by schema **S** is 1-colorable, then procedure PLACEPROPS finds a perfect 1-directional object layout. Otherwise, if the conflict graph is 2-colorable, then the procedure finds a perfect 2-directional object layout.

Proof. If the conflict graph G induced by **S** is 1-colorable, then there are no edges, and therefore $x \rightleftharpoons y$ does not hold for any pair of properties x and y in *props*. In this case, Step 2 must then assign $Dir(x)$ the value 1 for all properties x in *props*. Since the procedure assigns offsets according to Algorithm A, it finds a perfect 1-directional object layout by Theorem 1.

If the conflict graph is not 1-colorable but 2-colorable, then we prove by contradiction that $frustrated = \emptyset$ for all iterations of the main loop, and therefore that the procedure assigns directions to properties according to a valid 2-coloring of G.

Let x_0 denote the first property chosen for placement in which $x_0 \in frustrated$. Then there are fields x_1 and y_1 in *placed* such that $Dir(x_1) \neq Dir(y_1)$ and both $x_0 \rightleftharpoons x_1$ and $x_0 \rightleftharpoons y_1$ are true. Let $(x_1, x_2, \ldots, x_l)$ denote a maximal length sequence of properties in *placed* such that $x_i \rightleftharpoons x_{i+1}$ holds for all $1 \leq i < l$, and such that later properties are placed by the procedure before earlier properties. Let $(y_1, y_2, \ldots, y_m)$ denote likewise. If $x_i = y_j$, for some $1 \leq i \leq l$ and $1 \leq j \leq m$, then the conflict graph G must contain an odd cycle (by virtue of the choice of x_0), and therefore would not be 2-colorable, contrary to assumptions. Without loss of generality, assume property y_m is placed by the procedure after property x_l. Then there exists $0 \leq i < l$ such that property x_i is placed after y_m and property x_{i+1} is placed before. Since $y_m \rightleftharpoons z$ is not true for any $z \in placed$ at the time y_m is chosen for placement, $y_m \notin (desperate \cup frustrated)$ at this time, and therefore $(desperate \cup frustrated) = \emptyset$. But then x_i must occur in *unplaced*, and then $x_i \rightleftharpoons x_{i+1}$ cannot hold by Lemma 2—a contradiction. □

Corollary 1: If at most one immediate superclass is mentioned in any class definition occurring in a given type schema **S**, then procedure PLACEPROPS computes a perfect 1-directional object layout for **S**.

Proof. A property x mentioned in the definition of a class C implies all properties mentioned in the definition of any superclasses, and is implied by all properties mentioned in the definition of any subclasses. Since the generalization taxonomy in this case has the shape of a forest of trees, $Jcnt(x, y) = 0$ for any property y occurring in the definition of a class that is neither a subclass nor a superclass of C. Therefore, there are no properties x and y such that $x \rightleftharpoons y$ and the conflict graph for **S** contains no edges. □

The problem outlined above, in which a type schema may have a perfect 1 or 2-directional object layout but have a conflict graph that is not 2-colorable, relates directly to virtual class definitions. In particular, we show in the following that perfect n-directional object layouts imply n-colorings when none of the classes defined in a given type schema are declared as virtual.

Definition 5: A type schema is *simple* if and only if there are no virtual class definitions.

Theorem 5: If a perfect n-directional object layout exists for a simple type schema **S**, then the conflict graph $G = (props, E)$ induced by **S** is n-colorable.

Proof. Assume we are given a perfect n-directional object layout for **S**. The theorem follows if we can prove $\{x, y\} \notin E$ for any pair of properties x and y for which $Dir(x) = Dir(y)$. For any such pair of properties, there are two cases to consider.

Case 1, where $Off(x) = Off(y)$. In this case, $Jcnt(x, y) = 0$ since any perfect layout is also a legal layout by definition, and therefore $\{x, y\} \notin E$ also by definition.

Case 2, where $Off(x) \neq Off(y)$. Assume without loss of generality that $Off(x) < Off(y)$. We prove that either (1) $x \Rightarrow y$ is true , (2) $y \Rightarrow x$ is true, or that (3) $Jcnt(x, y) = 0$. (If any of these conditions hold, then $\{x, y\} \notin E$ by definition, and the theorem follows.)

Let C be the unique class mentioning y in its definition. Since **S** is simple by assumption, C must be non-virtual and must therefore have a property set in the property set list for **S**. Also, since we are given a perfect layout, then by a simple induction on $Off(y) - Off(x)$, there exists a property z in $Pset(C)$ (not necessarily distinct from x) for which $Dir(z) = Dir(x)$ and $Off(z) = Off(x)$. Therefore, z must occur together with y in any property sets for subclasses of C; that is,

$$Jcnt(z, y) = Rcnt(y). \tag{4.2}$$

This implies either that $z \Rightarrow y$ is true or that $y \Rightarrow z$ is true. If $z = x$, then either condition (1) or condition (2) above holds. Otherwise, if $z \neq x$, $Jcnt(x, z) = 0$, again since any perfect layout is also a legal layout. But this fact together with (4.2) then implies condition (3) above by Lemma 1. □

Corollary 2: Procedure PLACEPROPS finds a perfect 1 or 2-directional object layout for a simple class schema **S** if one exists.

Proof. The corollary follows immediately from the correctness of PLACEPROPS (Theorem 3) and from Theorems 4 and 5 above. □

4.3 Some empirical results.

We conducted a number of experiments with procedure PLACEPROPS together with the C++ and the conversion to single inheritance methods reviewed earlier. In the first set of experiments, we used these approaches to derive object layouts for class schema for a C++ graphics spline library and for a Lisp Flavors system. The results of these experiments are reported in Table 6 (the "conversion to single inheritance" method is referred to as the "tree method" in the figure). The data indicates the percentage of wasted store; that is, the percentage of store allocated to objects that is *not* used for encoding property values. In calculating this data for both sets of experiments, we assumed a uniform distribution of objects to classes, and also that all properties are pointer sized. (The latter is true in the case of the Flavors data and is nearly true in the case of the spline library.) The result of an independent experiment using procedure PLACEPROPS with $n = 2$ on a CLOS class hierarchy [1] is also included in the table.

Table 6: Experiments on Real Class Schemas.

	# of classes	# of properties	PLACEPROPS $n = 2$	C++ METHOD	TREE METHOD
Graphics Spline Library	276	826	0%	3%	10%
Lisp Flavors System	564	2245	6%	10%	39%
CLOS Class Hierarchy	≈750	≈1000	7%	(n/a)	(n/a)

In the second set of experiments, we evaluated the performance of these approaches on a number of random class schema. A triplet of numbers describes each test scenario: *levels*×*count*×*extra*. We assume each class includes a single property is its definition. The first component of a given scenario, *level*, specifies the number of levels occurring in the class hierarchy. The second component, *count*, specifies the number of class definitions generated for each level in the hierarchy. For all levels i greater than 1, each class inherits from a random class at level $i - 1$. In addition, *extra* random inheritance links are added from level i to level $i - 1$.

Figure 5 illustrates one of the randomly generated class schemas for the scenario $3 \times 4 \times 1$ that was used in this second set of experiments. In this figure, each vertex represents a class definition with the class property and superclasses given by the vertex label and outgoing arcs respectively. Thus, a new object for the class defining property j would have additional values for properties c and h. Figure 6 illustrates the layouts produced for this schema by most of the methods mentioned in Table 7 below. A location in an object layout that is labeled "$\star$" indicates wasted store. Thus, procedure PLACEPROPS found a perfect 2-directional layout and obtained a 1-directional layout with three units of wasted store. Four units of wasted store were required by the tree method. For the C++ method, a total of 5 units of extra store were required to ensure that class l would have a single copy of property c.

The results of this second set of experiments are reported in Table 7. All data given in the

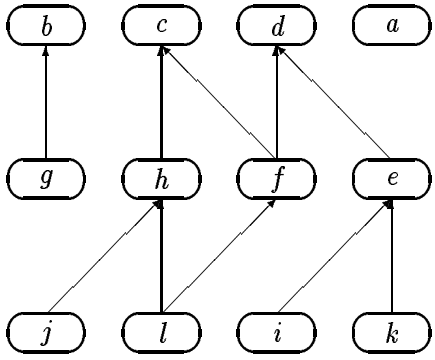


Figure 5: A RANDOMLY GENERATED CLASS SCHEMA (SCENARIO 3×4×1)

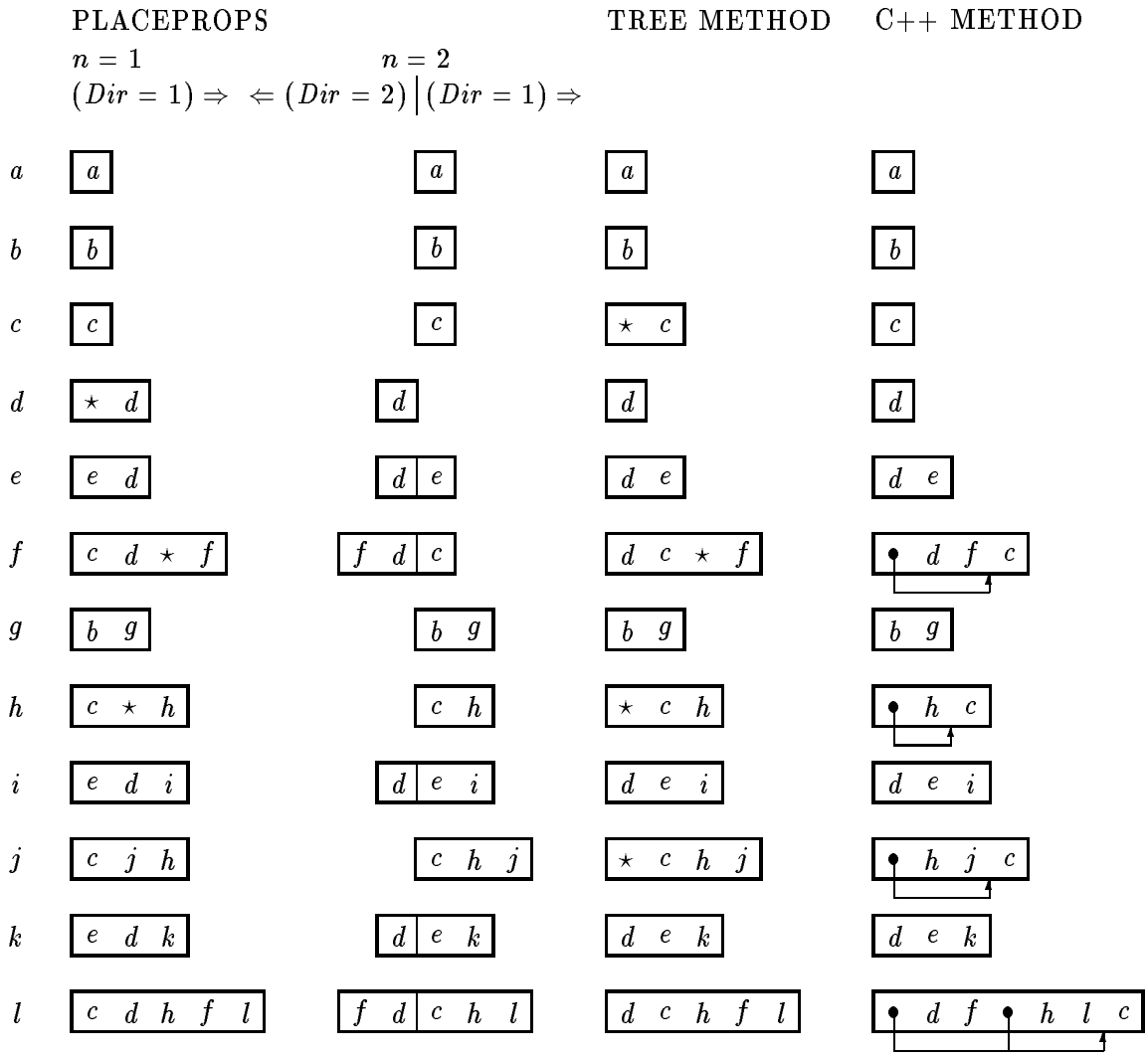


Figure 6: LAYOUTS PRODUCED BY VARIOUS METHODS FOR FIGURE 4.

table is an average over five schemas generated according to the indicated scenario. All data outside of parentheses indicates a percentage of wasted store. The data given in parentheses for the case of procedure PLACEPROPS invoked with $n > 2$ indicates the percentage of properties that must be placed in a direction greater than 2 in order to ensure that there is no wasted store. Thus, with the scenario $5 \times 16 \times 2$ for example, the data indicates that extracting values for 4% of all properties would require a more expensive table lookup.

Table 7: Experiments on Randomly Generated Class Schemas.

	PLACEPROPS				
scenario	$n = 1$	$n = 2$	$n > 2$	C++ METHOD	TREE METHOD
$3 \times 4 \times 1$	15%	1%	(3%)	6%	19%
$4 \times 6 \times 1$	12%	<1%	(1%)	11%	19%
$4 \times 6 \times 2$	30%	13%	(16%)	15%	39%
$5 \times 16 \times 1$	15%	1%	(1%)	1%	25%
$5 \times 16 \times 2$	23%	6%	(4%)	6%	34%
$5 \times 16 \times 3$	39%	20%	(9%)	9%	46%
$5 \times 16 \times 4$	40%	21%	(16%)	8%	60%

In these experiments, the only circumstances in which the 2-directional layouts obtained by procedure PLACEPROPS were not one of the best overall occurred in the case of randomly generated class schema for the scenarios $5 \times 16 \times 3$ and $5 \times 16 \times 4$. In these cases, the C++ approach required between a third and a half of the store overhead of the 2-directional layouts on average. However, the reverse was true for two of the non-random class schemas based on real data! Overall, the experiments indicate that feasible 2-directional layouts are likely to be obtained by procedure PLACEPROPS for class schema occurring in practice, and that one can expect the percentage of wasted store required for such layouts to be less than for layouts obtained by existing methods.

One of our implementations in C of procedure PLACEPROPS can obtain a 2-directional layout at the rate of 1000 properties every 6 seconds when running on a Decstation 3100. As written, the worst case running time for this implementation is $O(|props|^3)$. This can be reduced to $O(|props|^2)$ by refining the implementation with techniques such as incrementally maintaining the desperate and frustrated sets, as opposed to recomputing the sets from scratch after each field is placed (although the details of this are beyond the scope of this paper). However, in view of the performance of this straightforward implementation, such optimizations do not appear to be necessary in practice.

4.4 On the difficulty of optimal n-directional object layouts.

Note that Theorem 4 and Corollary 2 do not prove any results about the ability of procedure PLACEPROPS to perform merging well, that is, to minimize the need for wasted store in cases where the conflict graph induced by a given schema is not 2-colorable (although our experimental evidence suggests that the procedure works very well in practice). Indeed, in this subsection, we prove

that it would be pointless to search for an optimal n-directional layout algorithm. In particular, we prove that determining if an arbitrary type schema has a perfect n-directional object layout is NP-complete for $n > 0$, and that this remains true for simple type schema for $n > 2$. Since an algorithm that finds an optimal layout could decide if a perfect layout existed, finding optimal n-directional object layouts is NP-hard.

Theorem 6: The problem of determining whether or not a simple type schema has a perfect n-directional object layout is NP-complete for $n > 2$.

Proof. For an arbitrary graph $G' = (V', E')$, we can construct a simple type schema **S** inducing a conflict graph $G = (\mathit{props}, E)$ isomorphic to G' as follows: for each vertex $x \in V'$ add the class definition

$$C_x = \text{CLASS ()}\ x\text{: INT; END},$$

and for each edge $\{x, y\} \in E'$ add the "edge" class definition

$$C_{xy} = \text{CLASS } (C_x, C_y)\ \text{END}.$$

Clearly, $x \in \mathit{props}$ if and only if and $x \in V'$. If $\{x, y\} \in E'$, then $\mathit{Jcnt}(x, y) = 1$ and both $\mathit{Rcnt}(x)$ and $\mathit{Rcnt}(y)$ exceed 1. Thus, neither $x \Rightarrow y$ nor $y \Rightarrow x$ is true and therefore $\{x, y\} \in E$. Conversely, if $\{x, y\} \in E$, then $\mathit{Jcnt}(x, y) > 0$ and there exists an edge class definition of the form above, which implies $\{x, y\} \in E'$. Thus, G must be isomorphic to G' and therefore, by Theorem 1 and Theorem 5, **S** has a perfect n-directional layout if and only if G' is n-colorable. The theorem follows since we can easily check a layout to see if it is perfect, and since the problem of determining if an arbitrary graph is n-colorable is NP-complete for $n > 2$ [5]. □

Theorem 7: The problem of determining whether or not an arbitrary type schema has a perfect n-directional object layout is NP-complete for $n > 0$.

Proof. The case for $n > 2$ follows from Theorem 6. Our proof of the remaining cases is again by reduction of the graph coloring problem, and is based on our proofs of Theorem 2 and Theorem 6.

First consider where $n = 1$. Given an arbitrary graph $G' = (V', E')$ and integer k, construct a type schema **S** as follows: for each vertex $x \in V'$ add the class definition

$$C_x = \text{VIRTUAL CLASS ()}\ x\text{: INT; END},$$

and for each edge $\{x, y\} \in E'$ add the class definition

$$C_{xy} = \text{CLASS } (C_x, C_y)\ xy_3\text{: INT; } \ldots; xy_k\text{: INT; END}.$$

Thus, the property set list for **S** consists of a property set for each edge $\{x, y\} \in G'$ of the form $\{x, y, xy_3, \ldots, xy_k\}$. If G' is k-colorable, then a perfect 1-directional object layout is obtained as outlined in the prove to Theorem 2. Conversely, if we are given a perfect 1-directional object layout for **S**, then this layout gives a k-coloring for G': if vertex x in V' is assigned offset i, then assign x

the color i; otherwise, if vertex x is not assigned an offset, then it has no incident edges and therefore assign it the color 1.

Now consider where $n = 2$. For an arbitrary graph $G' = (V', E')$ and integer k, construct a type schema **S** as follows: first, initialize **S** with the following pair of "boundary" classes:

BlockLeft = VIRTUAL CLASS () *bl*: INT; END, and
BlockRight = VIRTUAL CLASS () *br*: INT; END.

Now add vertex classes to **S** as in the case for $n = 1$ above, and for each edge $\{x, y\} \in G'$ add *four* class definitions of the form

C_{xy} = VIRTUAL CLASS (C_x, C_y) xy_3: INT; ...; xy_k: INT; END,
C_{xy}^{bl} = CLASS (BlockLeft, C_{xy}) END,
C_{xy}^{br} = CLASS (C_{xy}, BlockRight) END and
C_{xy}^{blr} = CLASS (C_{xy}^{bl}, C_{xy}^{br}) END.

In this case, each edge $\{x, y\} \in G'$ contributes the three property sets

$$Pset(C_{xy}^{bl}) = \{bl, x, y, xy_3, \ldots xy_k\},$$
$$Pset(C_{xy}^{br}) = \{x, y, xy_3, \ldots xy_k, br\} \text{ and}$$
$$Pset(C_{xy}^{blr}) = \{bl, x, y, xy_3, \ldots xy_k, br\}$$

to the property set list for **S**. In any legal 1-directional layout, either $Pset(C_{xy}^{bl})$ or $Pset(C_{xy}^{br})$ must have a "hole" occurring either for property *br* or property *bl* respectively. Thus, a perfect layout must be at least 2-directional with *bl* and *br* assigned different directions. Assume, without loss of generality, that $Dir(br) = 1$ and that $Dir(bl) = 2$. If the layout is perfect, then *br* must be assigned the maximum offset of all properties assigned direction 1 (otherwise, $Pset(C_{xy}^{bl})$ would again have a hole). This holds analogously for property *bl* with respect to $Pset(C_{xy}^{br})$. But then $Pset(C_{xy}^{blr})$ implies that the remaining k properties must be assigned all remaining offsets prior to those assigned *bl* and *br*. This gives a k-coloring for G': if vertex x in V' is assigned direction 1, then assign x the color $Off(x)$; if vertex x is assigned direction 2, then assign x the color $Off(x) + Off(br) - 1$; otherwise, if vertex x is not assigned a direction, then it has no incident edges and therefore assign it the color 1. Similarly, if there is a k-coloring for G', then a perfect 2-directional object layout is again obtained as outlined in the prove to Theorem 2. □

4.5 Generating record definitions.

After computing a 1 or 2-directional object layout for a given type schema **S**, generating a record type in our particular type language that specifies an object encoding for a non-virtual class C in **S** is straightforward. A procedure for doing this is given in Figure 7. In this case, fields encoding the values for any properties in $Pset(C)$ placed in the second direction will occur in the record type for C in descending order of their offset, and are followed by fields encoding values for any remaining properties placed in the first direction in ascending order of their offset. Note that the location of fields in record types for distinct classes that encode values for the same property in *props* remains constant. In particular, let k denote the maximum offset of all properties in *props* placed in the

second direction, and assume the ith field in the record type for C encodes values for property x in $Pset(C)$. Then either $Dir(x) = 2$ and $i = k - Off(x) + 1$, or $Dir(x) = 1$ and $i = k + Off(x)$.

This ith field can directly encode values for x if x qualifies as an *inline* property; otherwise, the field must point indirectly to such values. For our particular type language, the circumstances in which this level of indirection is not needed are as follows:

1. whenever x is of type INT or POINTER (i.e. of unit size), or

2. whenever x is placed in the first direction and there does not exist another property y in $Pset(C)$ that conflicts with x and is placed in the same direction as x but at a greater offset.

For type languages that allow one to reason more extensively about the actual internal encoding of record types, the second of these conditions can be generalized by removing the restriction that x is placed in the first direction. In our case, we have assumed only two things: that INTs are allocated the same amount of store as POINTERs, and that space for fields is allocated in the order in which the fields occur in a record type.

For an example of a case where property indirection is needed, consider the university schema with the definition of class Teacher given instead by

Teacher = CLASS (Person) Address: STRING(40); END.

Also assume procedure PLACEPROPS has computed the following 1-directional property layout for the new schema.

$$\begin{array}{llll} Dir(\text{Name}) = 1 & Dir(\text{Address}) = 1 & Dir(\text{Advisor}) = 1 & Dir(\text{Boss}) = 1 \\ Off(\text{Name}) = 1 & Off(\text{Address}) = 2 & Off(\text{Advisor}) = 3 & Off(\text{Boss}) = 4 \end{array}$$

The object encoding computed by procedure GENRECDEFS for this case is listed in Table 8. Note that, since the Address property fails to satisfy either of the above two conditions, a level of indirection in its encoding is needed within the record types for classes Teacher and Tutor. To further clarify this, Figure 8 illustrates corresponding instances of the records in Figure 2 (but with values for the Address property replacing values for the Salary property).

5. Summary and Discussion

We have presented a new algorithm for obtaining an object encoding for object-oriented programming languages that allow subtyping and multiple inheritance among class definitions. The new algorithm incorporates a refined version of Algorithm B in [9] and may be easily adapted for use in a preprocessor for existing languages. Our experiments indicate that, for class schema occurring in practice, the algorithm is able to produce an object encoding with minimal store overhead in which property value access for pointer-sized properties can proceed with the same efficiency as in systems that do not provide multiple inheritance. For some of the properties that require a larger amount of store, a single level of indirection is needed in the worst case. With our algorithm, a simple comparison can always be used to check for equality among object references, and no overhead whatever is needed

procedure GENRECDEFS(**S**, *Dir*, *Off*)

(initialization)

$k := \max(\{0\} \cup \{\ 1 \le j \mid \exists x \in \mathit{props}$ s.t. $\mathit{Dir}(x) = 2$ and $\mathit{Off}(x) = j\ \})$;
for each $x \in \mathit{props}$,
    $\mathit{Loc}(x) := ((\mathit{Dir}(x) - 1) \cdot (k - \mathit{Off}(x) + 1)) + ((2 - \mathit{Dir}(x)) \cdot (k + \mathit{Off}(x)))$;
$\mathit{inline} := \{\ x \in \mathit{props} \mid x$ is declared of type INT or POINTER in **S** $\}$;
$\mathit{inline} := \mathit{inline} \cup \{\ x \in \mathit{props} \mid \mathit{Dir}(x) = 1$ and $\forall y \in \mathit{props}$,
        $(\mathit{Dir}(y) = 2$ or $\mathit{Off}(y) \le \mathit{Off}(x)$ or $y \Rightarrow x$ or $\mathit{Jcnt}(y, x) = 0)\ \}$;
for each $x \in (\mathit{props} - \mathit{inline})$, generate "$x$Template = RECORD x: $\mathit{Type}(x)$; END";

(loop body—do for each non-virtual class C in S)

generate "C = RECORD";
for $i := 1$ to $(k + \max(\{0\} \cup \{\ 1 \le j \mid \exists x \in \mathit{Pset}(C)$ s.t. $\mathit{Dir}(x) = 1$ and $\mathit{Off}(x) = j\ \}))$,
    if there exists $x \in \mathit{Pset}(C)$ s.t. $\mathit{Loc}(x) = i$
        then if $x \in \mathit{inline}$
            then generate "x: $\mathit{Type}(x)$;";
            else generate "x: POINTER TO xTemplate;";
        else if there exists $y \in \mathit{Pset}(C)$ s.t. $\mathit{Loc}(y) < i$
            then generate "SkipOff$i$: INT;";
            else generate "ShiftOff$i$: INT;";
for each $x \in \mathit{Pset}(C) - \mathit{inline}$, generate "$x$Store: xTemplate;";
generate "END";

Figure 7: A PROCEDURE FOR GENERATING AN OBJECT ENCODING.

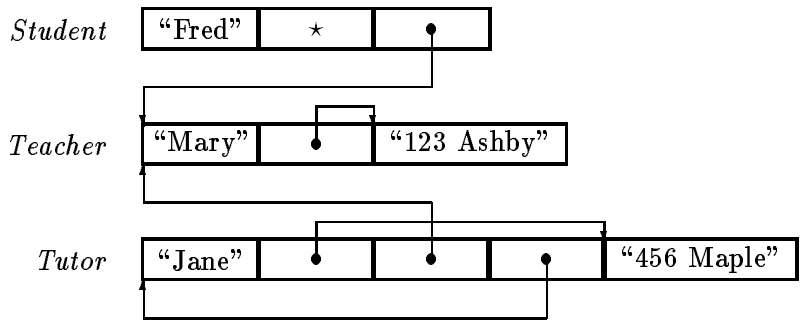


Figure 8: EXAMPLE 1-DIRECTIONAL OBJECT LAYOUT WITH FIELD INDIRECTION.

Table 8: 1-DIRECTIONAL OBJECT LAYOUT WITH FIELD INDIRECTION.

```
AddressTemplate = RECORD Address: STRING(40); END

Student = RECORD
    Name: STRING(20);
    SkipOff2: INT;
    Advisor: POINTER TO Teacher;
END

Teacher = RECORD
    Name: STRING(20);
    Address: POINTER TO AddressTemplate;
    AddressStore: AddressTemplate;
END

Tutor = RECORD
    Name: STRING(20);
    Address: POINTER TO AddressTemplate;
    Advisor: POINTER TO Teacher;
    Boss: POINTER TO Teacher;
    AddressStore: AddressTemplate;
END
```

for type coercions (including coercions that cannot be guaranteed safe at compile time). Also, the algorithm can be used to implement method dispatch when applied to the problem of finding an object encoding for class objects. (In this case, property values correspond to function addresses.)

However, our algorithm leaves open a number of issues. In the remainder of this section, we comment on several of these issues, and in the process suggest a number of directions for future research.

5.1 Non-uniform object distributions.

Our procedure for computing an object layout and our experimental evaluation of this procedure assume that objects are distributed uniformly among classes—that objects will be created in different classes at about the same rate. However, in some situations, we might have object count estimates available that give a more accurate model of the actual distribution of objects among classes at run time. For example, in the case of the university schema, a more accurate model would indicate that

the number of Student objects will typically be much greater than the number of either Teacher or Tutor objects. If object count estimates are supplied in the form of a weight attached to each class type in the input schema, then it is straightforward to incorporate such estimates in our *Rcnt* and *Jcnt* data computed during the initial schema analysis phase. In this new definition, $Rcnt(x)$, for example, should be computed as the sum of the weights attached to the classes that can have objects with values for property x. (Our uniformity assumptions are then a special case of this more general setting, in which a weight of 0 is assumed for virtual classes and a weight of 1 for non-virtual classes.)

A number of preliminary experiments have been conducted on the tree method for random schema that made use of object count estimates [11]. The experiments considered the effect of increasing the relative number of objects occurring in classes lower in the class hierarchy, and the effect of non-uniform "two-step" object distributions among classes at the same level in this hierarchy. In both circumstances, the performance of the tree method improved considerably. We believe this will also be the case for our multidirectional algorithm when class weights are factored into the computation of *Rcnt* and *Jcnt* data in the manner suggested. Note that, in contrast, the first circumstance will clearly *decrease* the performance of the C++ method since objects in classes lower in the taxonomy are more likely to require additional store for what we referred to as "special fields" in our review section.

5.2 On varying sized fields.

As presented, our algorithm works best for cases in which all properties are pointer-sized. This is usually true for polymorphic programming languages, since in such languages each value is either a pointer or a pointer-sized integer. We believe that the single level of indirection that may be needed for properties which require a larger amount of store (e.g. properties that are arrays or strings) will not significantly impact performance. The reason is that the cost of this indirection will be amortized over what is likely to be a much greater amount of computation on the value itself.

Some refinements to the algorithm are necessary, however, to properly handle cases in which properties are less than pointer-sized (e.g. properties that are Boolean valued). A simple approach might be to preprocess a given type schema to "bin pack" groups of such properties into pointer-sized blocks. These blocks would then be placed as a unit by procedure PLACEPROPS. However, further experimentation with this approach requires a larger collection of real-world class schemas than is presently available to us. Clearly, the problem of handling varying sized fields merits some further study.

5.3 On interactive use and separate compilation.

If our algorithm is used in a preprocessor for an existing language, then it may become necessary to recompile a large part of a software system when certain kinds of changes are made to a global type schema. In particular, adding a new class type that multiply inherits from existing class types or modifying the definition of an existing class type may have this effect. This is because such changes may conflict with previous decisions about which fields to merge. However, this is not true in general. It is straightforward to derive an incremental version of our algorithm that does not require us to

forego separate compilation when adding new properties to the definition of existing classes, or when adding new class types that do not multiply inherit from existing class types.

Another more ambitious approach is to apply our algorithm at the time a set of object code files are linked to produce an executable file. This approach will require store to be allocated in object code files for "$\rightarrow$" operators, and then loaded with appropriate machine code at link time. Since the running time of our algorithm on a global schema is likely to not be worse than the time to compile, say, a new module containing a few hundreds lines of source code, such an approach seems feasible.

The situation with current methods is not much better. With the C++ method, one can dynamically add new class types without restriction only if superclasses are always declared virtual. This adds significantly to the storage overhead for special fields (at least a 21% overhead in the case of both the graphics spline library and Lisp Flavors system), and also adds to the overhead of accessing the values of inherited properties. Another approach used by Cardelli in his Quest language [3] is to introduce a second class type that does not allow multiple inheritance. But this seems to encourage the writing of programs that do not have class types allowing multiple inheritance for efficiency reasons, creating libraries that cannot be easily adapted to multiple inheritance later.

5.4 Inline classes.

Consider a simple modification to our type definition language to allow *inline classes*; that is, to allow a property to be declared of type "C" (instead of "POINTER TO C"). The reasons for allowing this relate to performance. For example, replacing the definition of the Student class in the university schema with the definition

Student = CLASS (Person) Advisor: Teacher; END

would have the effect of duplicating Teacher property values in Student records. This would enable both"$\rightarrow$" operators in the expression

$$\mathit{student} \rightarrow \text{Advisor} \rightarrow \text{Name}$$

to be compiled as a single machine-level indexed load instruction.

This particular case should perhaps not be allowed because of the possibility of a cycle (we may have two tutors who advise each other). However, the idea of inline classes still seems worthwhile for cases in which a cycle is not a possibility. Note that no revisions are necessary to our algorithm to handle non-cyclic cases, and that the above optimization on a sequence of "$\rightarrow$" operators can be applied whenever indirection is not needed for any of the properties mentioned in the sequence. However, it can be argued that an object encoding that disallows any such optimizations violates the intentions of the programmer. Removing the need for property indirection by our algorithm would overcome this problem, and is a topic for future research.

References

[1] K. A. Barret, 1990. Personal communication.

[2] A. H. Borning and D. H. H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proc. AAAI National Conference on Artificial Intelligence*, pages 234–238, 1982.

[3] L. Cardelli. Typeful programming. Technical Report 46, DEC SRC, May 1989.

[4] G. V. Cormack, 1990. Personal communication.

[5] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP–complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[6] D. E. Knuth. *The Art of Computer Programming; Volume 1.* Addison-Wesley, 1968.

[7] S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25:318–326, 1985.

[8] S. T. March. Techniques for structuring database records. *ACM Computing Surveys*, 15(1):45–79, March 1983.

[9] W. Pugh and G. E. Weddell. Two-directional record layout for multiple inheritance. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 85–91, June 1990.

[10] B. Stroustrup. Multiple inheritance in C++. In *Proc. EUUG Conference*, pages 1–17, May 1987.

[11] G. E. Weddell. Physical design and query compilation for a semantic data model (assuming memory residence). Technical Report CSRI-198, Computer Systems Research Insitute, University of Toronto, April 1987.

[12] G. E. Weddell. Efficient property access in memory-resident object oriented databases. Research Report CS-89-49, Department of Computer Science, University of Waterloo, 1989.

[13] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):202–214, April 1988.