View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by St Andrews Research Repository

This paper should be referenced as:

Connor, R.C.H., Dearle, A., Morrison, R. & Brown, A.L. “An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance”. In Proc. OOPSLA'89, New Orleans (1989).

An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance

Connor R.C.H., Dearle A, Morrison R. & Brown A.L.

Department of Computational Science
University of St Andrews
North Haugh
St Andrews
Fife
Scotland
KY16 9SS

richard%uk.ac.st-and.cs@ukc
al%uk.ac.st-and.cs@ukc
ron%uk.ac.st-and.cs@ukc
ab%uk.ac.st-and.cs@ukc

+44 334 76161 x8121

Abstract

In this paper we are concerned with addressing techniques for statically typed languages with multiple inheritance. The addressing techniques are responsible for the efficient implementation of record field selection. In object-oriented languages, this record selection is equivalent to the access of methods. Thus, the efficiency of these techniques greatly affects the overall performance of an object-oriented language. We will demonstrate that addresses, in such systems, cannot always be calculated statically and show how symbol tables have been used as address maps at run time. The essence of the paper is a new addressing technique that can statically calculate either the address of a field or the address of the address of the field. This technique is powerful enough to support an efficient implementation of multiple inheritance with implicit subtyping as described by Cardelli.

Keywords

addressing
multiple inheritance
static type checking
implicit subtyping

1 Introduction

Cardelli [car84] has given a semantics for multiple inheritance in statically typed languages with structural type equivalence. A type is a subtype of another if all operations allowed on the second type are also allowed on the first. In this paper we will concentrate on the record constructor, since we are concerned with addressing for field access on records; this is equivalent to method selection in object-oriented languages. Record types may have a subtype relation placed on them according to the selection operations defined on them. For example, we may define the types:

```
type thing is ( age : int )
type animal is ( age : int, food : string )
type leggedAnimal is ( age : int, food : string, noOfLegs : int )
```

In this example, *animal* is a subtype of *thing* and *leggedAnimal* is a subtype of *animal* and *thing*. Cardelli defines a record type τ to be a subtype (written $\leq$) of type τ' if τ has all the fields of τ', possibly some more, and that the common fields of τ and τ' are in the $\leq$ relation. Multiple inheritance occurs when τ may be a subtype of two or more unrelated types.

Instances of objects of type *leggedAnimal* may be created by using the constructor function *leggedAnimal* as follows:

```
let aLeggedAnimal = leggedAnimal( 3, "marmalade", 32 )
```

A field of the record may be selected to obtain its age as follows:

```
aLeggedAnimal.age
```

Thus the selector *age* may be considered to be a function with the following type definition:

```
age: leggedAnimal -> int
```

However, this function may also be used to find the age of an *animal* or a *thing*. In general, a function which operates on an object of some type τ' may safely operate on an object of type τ provided that $\tau \leq \tau'$. The age function may be written as:

```
let age = proc[ t ≤ thing ]( a : t -> int ) ; a.age
```

This definition says that the function *age* may take as a parameter any object which is a subtype of *thing* and return an integer. Functions such as *age* exhibit bounded universal quantification which Cardelli shows to be equivalent to inclusion polymorphism and inheritance.

In this paper, we investigate techniques for addressing fields of records in languages with static type checking, multiple inheritance and structural type equivalence. We will show that, in such systems, field addresses cannot always be calculated statically. To overcome this, symbol tables may be used as address maps at run time to perform the correct address translation. Here we describe a new addressing technique that can calculate either the address of a field or the address of the address of the field statically. This technique is powerful enough to support the implementation of multiple inheritance with implicit subtyping as described by Cardelli.

As this paper is about implementation, the syntax of the programming examples is not carefully explained. No particular language has been used, but the constructs used have been chosen to make the meaning clear.

2 Implementation of Subtypes

Consider a function such as *age* defined above. This function may operate on an instance of any type which is a subtype of the type *thing*. In practice, this means that the *age* function must be able to determine the position of the *age* field in the record. This may be achieved in two ways: either the *age* field must be stored at the same offset in all instances of subtypes of *thing*; or some mechanism must be provided so that the location of the *age* field may be calculated. We will return to the second possibility later in the paper and for the moment only consider the first possibility.

Making the address for *age* be the same for all instances of subtypes of *thing* may be easily achieved by placing all the fields of subtypes of *thing* after fields common to both *thing* and the subtypes of *thing*. For example, consider the following object definitions:

```
let aThing = thing( 3)
let anAnimal =  animal( 3, "grass" )
let aLeggedAnimal = leggedAnimal( 3, "grass", 4 )
```

These definitions may be implemented as in Figure 2.1.

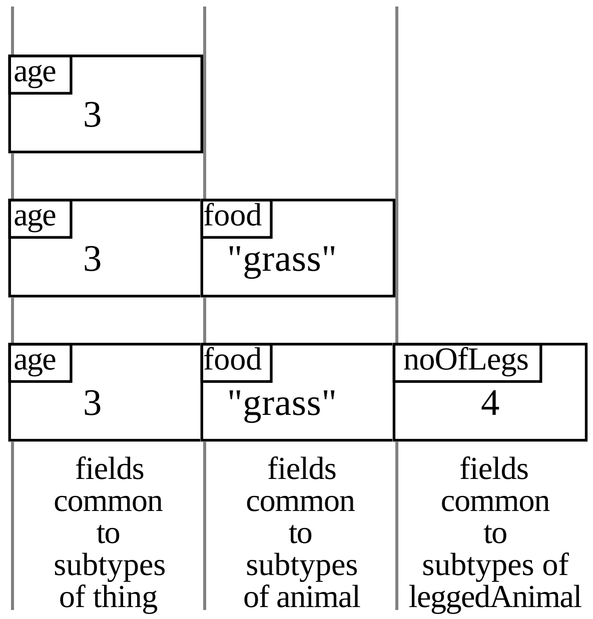
age
3
age
3
food
"grass"
age
3
food
"grass"
noOfLegs
4
fields common to subtypes of thing
fields common to subtypes of animal
fields common to subtypes of leggedAnimal

Figure 2.1 Scheme for Single Subtype Inheritance

Notice that in each of the instances, the fields of subtypes are always in the same position. Therefore in a field selection the location of that field is always known statically. For example,

```
X.food
```

can be compiled as the second field of X. This scheme is very efficient and highly successful in implementing languages with single subtype inheritance [GR83,str86]. However, the technique is impractical for multiple inheritance with implicit subtyping.

Consider the following set of type definitions:

type thing **is** (age : **int**)
type animal **is** (age : **int**, food : **string**)
type leggedThing **is** (age : **int**, noOfLegs : **int**)
type leggedAnimal **is** (age : **int**, food : **string**, noOfLegs : **int**)

and the object declarations:

let fish = animal(2, "coral")
let myDesk = leggedThing(3, 4)
let Al = leggedAnimal(29, "curry", 2)

If these objects are implemented in the manner described above the objects shown in Figure 2.2 will be created.

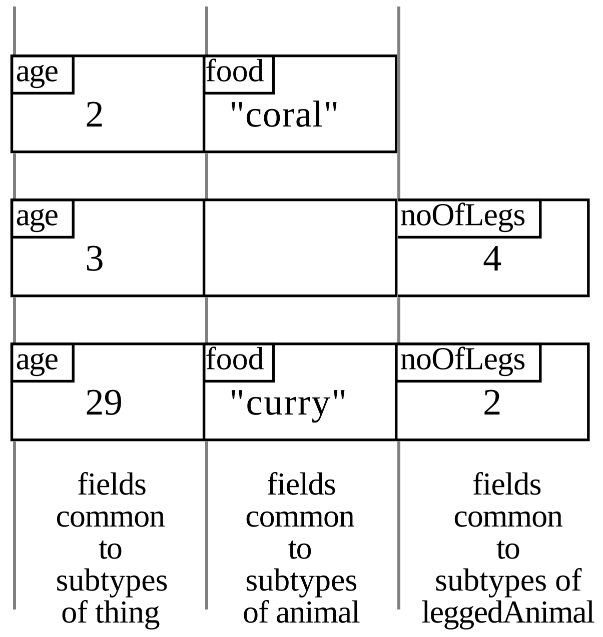
age
2
food
"coral"
age
3
noOfLegs
4
age
29
food
"curry"
noOfLegs
2
fields common to subtypes of thing
fields common to subtypes of animal
fields common to subtypes of leggedAnimal

Figure 2.2 A Multiple Inheritance Scheme

From the diagram it is immediately obvious that allocating fixed address slots to fields will leave gaps, and therefore waste space, in the records. For example, in the above, for *myDesk* to have its *noOfLegs* field in the correct place it has to leave a gap for the non-existent *food* field. This method is not viable as, for any type, there exist an infinite number of subtypes. This yields gaps of unbounded size in the objects. In practice, the size in any system is bounded by the number of concrete types, but this may be very large for long lived persistent systems [ABC83]. More importantly, if a new subtype is added to a system existing data must be restructured if it is to be compatible.

A variation of the scheme shown in Figure 2.1 has been proposed and implemented to support multiple inheritance with explicit subtyping in C++ [str87]. In this extension to C++, the order in which the components of a type are declared (inherited) determines the final order of the fields. Since the fields of a particular type occur as a single contiguous unit, the fields of each component supertype also occur as a single contiguous unit. This permits pointer arithmetic to be used to cast a pointer explicitly from a subtype to one of its component supertypes. The technique relies on an explicit field ordering enforced by the explicit multiple inheritance hierarchy. It is impractical for multiple inheritance with implicit subtyping: if new subtypes are introduced the field addressing must be recalculated, involving recompilation of all the components and restructuring of any persistent data.

2.1 Implementing Multiple Inheritance using Address Maps

In the single inheritance scheme discussed above, all field addressing is in the form of an offset from the base of the object. The scheme may be extended to work with multiple inheritance and implicit subtyping by using an indirection through an address map. The address map, which may be located at the start of every object, contains offsets for the fields belonging to that particular subtype. For example, the declaration

let Al = leggedAnimal(29, "curry", 2)

yields an object called *Al* . This object must have an address map containing the start positions of the fields *age*, *food*, and *noOfLegs*, as in Figure 2.3. Creating an object entails creating the address map as well as the fields of the record.

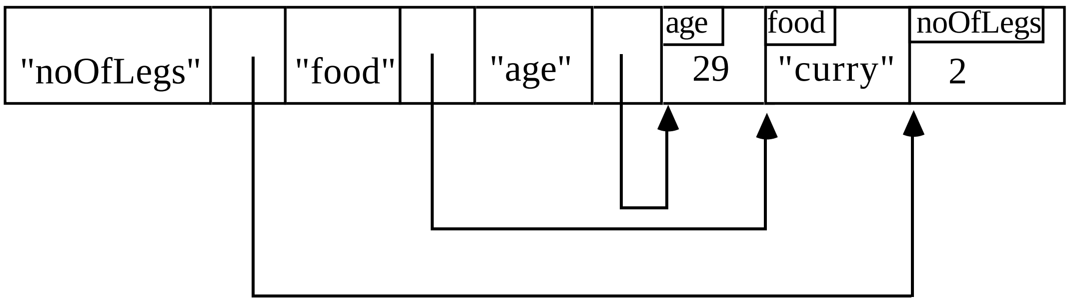

Figure 2.3 An Object with its own Address Map

For record selection, if X has a type ≤ animal then

X.food

can be compiled as the *food* field of X. This is resolved, at run time, by looking up the string "food" in the address map to yield the correct field offset.

An immediate optimisation of this technique is to have only one copy of the address map for every type of record, as in Figure 2.4. The record itself contains a pointer to the map. This arrangement may also have some advantages in identifying pointers for garbage collection.

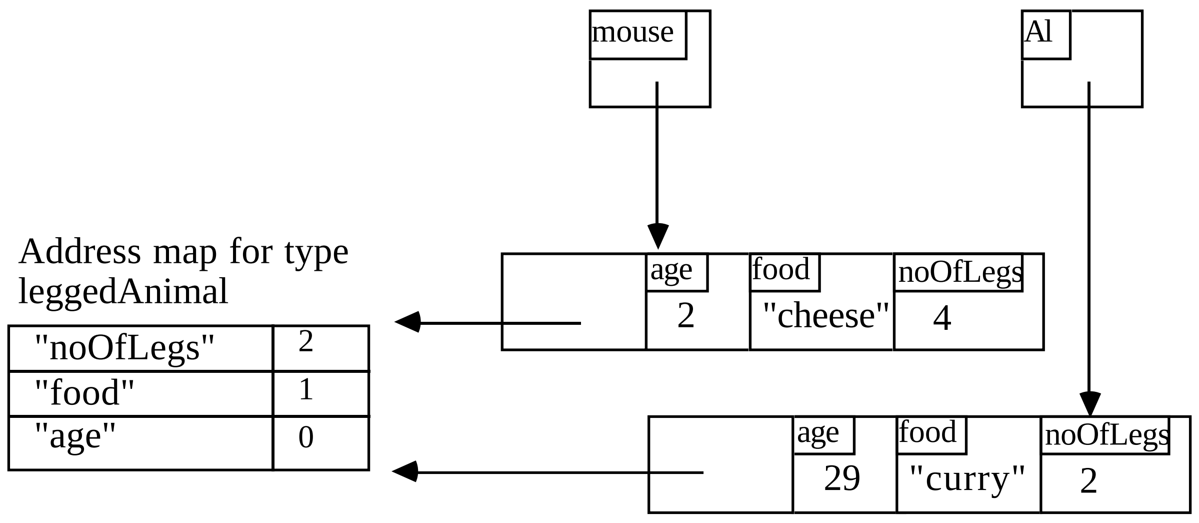

Figure 2.4 Sharing the Address Map

The string address maps will normally use a hashing function for lookup. They provide a very general solution for languages that allow the substitution of a subtype by a supertype [WZ88] and use structural equivalence as the type equivalence rule. The technique will also work for dynamically typed systems which are not discussed here.

A final advantage of the technique is that strings match across compilation units which means that there is no need for a centralised dictionary of field addresses. Each compilation unit can contain its own address map and still work consistently with independently prepared data. In object-oriented database systems [CL88,BBB88], and distributed systems, this aspect is a major consideration.

2.2 Variants of String Address Maps

An improvement in the efficiency of the address maps would be gained if they could be keyed by integer instead of string. To do this the compiler can keep a central dictionary of field names to which it can statically allocate a unique integer key to each field. Thus

```
X.food
```

can be translated, at compile time, to key (food) field of X. This key, which is an integer, can be used at run time to search the address map.

The drawback of this method is that it requires a centralised dictionary for field keys. In an object oriented database system, or a distributed system, these keys may become large and the dictionary holding the keys may constitute a bottleneck in the system. However, the technique may be suitable for systems which use name equivalence as their type equivalence rule, since the above problems are already present in such systems.

A further variation is found in the ObServer system [SZ86]. This is an object oriented database with name equivalence and explicit subtyping. That is, a type is a subtype of another only if it is declared to be so in the database schema. Consider the following schema:

type thing **is** (age : **int**)
type limbedThing **is** thing **with** (arms,legs : **int**)
type sightedLimbedThing **is** limbedThing **with** (eyes : **int**)

In the above, the fields may be grouped as: (age, (arms,legs), eyes). Thus the address map need only contain key/entries for these groups. The above schema may be used to create an object of type *sightedLimbedThing* by

let Ron = sightedLimbedThing(42,2,2,2)

Each set of common fields may be allocated an address ‘slot’. Within the slot (arms,legs), the field offsets of *arms* and *legs* can be calculated statically. (Figure 2.5)

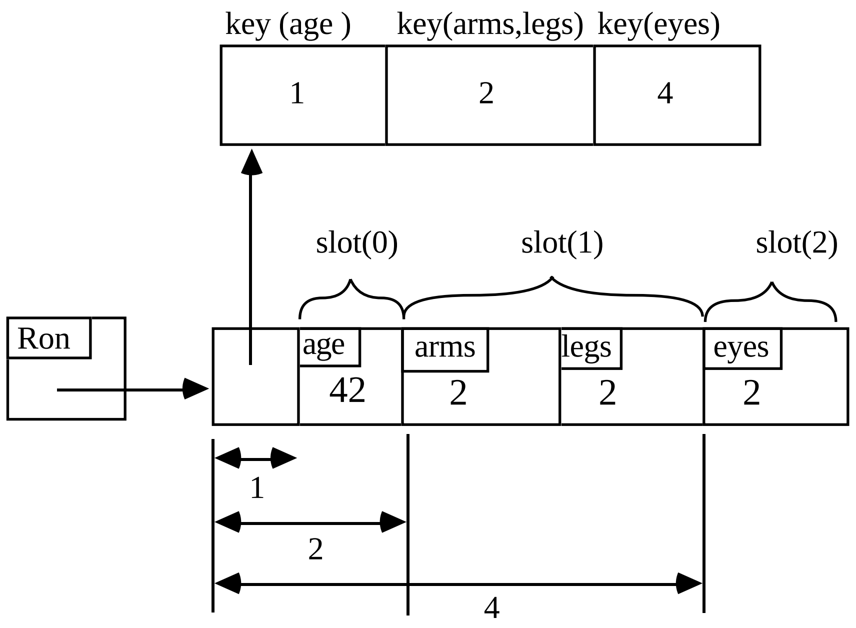

Figure 2.5 Fixed Slots for Subtypes

There are two advantages to this scheme. First, slots may be placed on different volumes, or distributed, an important consideration for large databases. Second, the addresses within the subtypes may be calculated statically. The disadvantage is that it works only for name equivalence with explicit subtyping, and may run into severe reorganisational overheads if the schema is edited to alter the subtype hierarchy.

We will now describe an implementation technique which uses an integer mapping for languages with structural type equivalence. Section 3 describes an implementation for languages which do not allow substitutability: that is, it is possible to tell statically the precise type of an object except where it is abstracted over explicitly by use of bounded universal quantification. Section 4 extends this implementation to allow for full substitutability.

3 Bounded Universal Quantification without Substitutability

If we consider the example of multiple inheritance given above, it is clear that there is an implicit function *noOfLegs* with which we may wish to find the number of legs belonging to either a table or a dog, or indeed any object which is a subtype of *leggedThing*. Perhaps we wish to write a procedure which tests whether an object with legs is stable. Such a procedure could look like this:

```
let fallsOver = proc[ t ≤ leggedThing ]( x : t -> bool )
        x.noOfLegs < 2
```

and could be used as:

```
let unsafe = fallsOver( myDesk )
let drunk = fallsOver( Al )
```

To determine the correct address fields, we propose a solution which uses information that is statically available at the call of such a procedure. In the above examples, at each procedure call the type of the operand is known statically, and the field offsets can be calculated. Calculating this information statically saves most of the cost of the associative lookup required with the lookup table solution.

Since the type of the procedure being called is known statically, it is possible to tell which fields of the operand object may be required during the execution of the procedure. For example, the procedure *fallsOver* is restricted to an operand of type *leggedThing*, and so only the *noOfLegs* and *age* fields may be required, no matter how many other fields the operand may contain.

Notice that it is only the type of the procedure which is required; if procedures are first-class values they may still be freely assigned.

The information may be encapsulated without altering the run-time support for the language being implemented, so long as this includes support for higher-order functions [AM85,BCC88,CBC89]. When the quantified procedure is compiled, it is compiled as two nested procedures, with the field offsets being parameters to the outer procedure and appearing as free variables within the inner procedure. The inner procedure, corresponding to the quantified one, is returned as the result of the outer. In the example,

```
let fallsOver = proc[ t ≤ leggedThing ]( x : t -> bool)
        x.noOfLegs < 2
```

it is known that whatever the type of the parameter, only the fields *age* and *noOfLegs* may be accessed. It is therefore compiled as if it were:

```
let fallsOver_wrapper = proc( age_offset,noOfLegs_offset : int -> proc( ? ->  bool ) )
        proc( x : ? -> bool )
                x( noOfLegs_offset ) < 2
```

At the point in the program where the procedure is called, the compiler can plant the integer values required as parameters to the wrapper procedure as literal values in the code stream, and no dynamic lookup is required. This is possible since the compiler may statically determine which offsets are required and the precise type of the operand. Thus:

```
fallsOver( myDesk )
fallsOver( Al )
```

is compiled as

```
( fallsOver_wrapper( 1,2 ) ) ( myDesk )
( fallsOver_wrapper( 1,3 ) ) ( Al )
```

and so when the *noOfLegs* field is accessed in the procedure, the second field of *myDesk* and the third field of *Al* will be looked up as required.

The type of the operand is not always known statically, but because of the generality of the solution no extra work is required for these cases. In the example

```
let atConference = proc[ t ≤ leggedAnimal ]( x : t -> bool )
        x.food = "curry" and fallsOver( x )
```

the type of the *x,* supplied to the *fallsOver* procedure as a parameter, is not known statically as it has already been abstracted over. It is clear, however, that *x* is a subtype of *leggedThing* and as such may be used as the operand of *fallsOver*. The compiler does not know statically the offsets it requires to pass to the *fallsOver* wrapper procedure, but it does know where they can be found, as they must be a subset of the offsets provided by the *atConference* wrapper procedure. The procedure then compiles as if it were:

```
let atConference_wrapper = proc( age_offset,food_offset,noOfLegs_offset -> proc( ? -> bool ) )
        proc( x : ? -> bool )
                x( food_offset ) = "curry" and
                ( fallsOver_wrapper( age_offset,noOfLegs_offset ) ) ( x )
```

and the correct values are introduced for the required offsets.

This technique of compiling higher-order functions has even greater advantage if the field lookups are performed many times with operands of the same type. When this happens the wrapper procedure is called only once, and once the free variables are in place no more work is needed to provide the correct offsets for use within the procedure. The following procedure incurs a slightly greater fixed cost, but has no penalty in proportion to the size of the array, when compared to the same procedure declared for only one of the allowed types.

```
let allFallOver = proc[ t ≤ leggedThing ]( a : array of t -> bool )
begin
        let res := true
        for i = 1 to max do
                if a[ i ].noOfLegs ≥ 2 do res := false
        res
end
```

If a procedure is going to be used many times with the same type of operand, the same efficiency can be achieved, by only calling the wrapper procedure once and using the returned value for all other instances of the call. Some programming languages provide syntax which would allow a programmer to specify this, by allowing explicit specialisation of a quantified procedure. Thus for example it may be possible to write,

```
let leggedAnimalFallsOver = fallsOver[ leggedAnimal ]
```

If this is allowed, then a quantified procedure which is called many times with the same type of operand may be written like this by the programmer, and the specialisation need only be performed once. It may be possible for a compiler to perform sufficient static analysis to notice when this may be advantageous.

4 Implementation of Substitutability

The above scheme relies upon the fact that the compiler knows the precise type of the object supplied as a parameter to a procedure application. If substitutability is allowed, this is no longer the case. We now extend the above solution to allow for this.

Substitutability allows a location of a particular type to have a value of any subtype assigned to it. For example, if a location is declared with a value of type *thing* assigned to it, then it may be updated with a value of type *animal*, as *animal* is a subtype of *thing*. The location continues to have type *thing*: the operation not only updates but also throws away type information, as the object may no longer be used as an *animal* but only as a *thing* from its new reference. For example:

```
let a := thing( 129 )
let b := animal( 23,"petrol" )
a := b
let d = a.age
```

is allowable, although

```
let e = a.food
```

is not. This is because the *food* field of the animal object can not be accessed through the location a, as this location is of type *thing*. Notice that it may still be accessed from the location b, as may the *age* field. The two locations have a different type view of the same object.

It is now no longer possible for the compiler to determine statically where to find the named fields of an object stored in a particular location, as this location may be updated with an object whose real type the compiler may not even know about at the time of compilation. Once again, addressing information must be planted.

It is obvious, however, that the solution already given must extend to this. The semantics of this assignment are similar to the replacement of the formal parameter of a quantified function by the actual parameter, with the corresponding type widening that occurs. This technique works by allocating space with the formal parameter location where the addressing information required may be accessed; similar space may be allocated instead with each record type location.

A straightforward way to achieve this is to implement a record type location with a double pointer, instead of a single one. Thus, we have one pointer which references the original record object, and another which points to an address map for the fields. This may appear superficially similar to a more conventional address map solution, but the following points should be noted:

- The address map is not a conceptual part of the object, but is associated semantically object may be viewed through a number of different address maps. (Figure 4.1)
- The map contains only the addresses within the object which may be accessed through that location, and has no information about any other fields which may be in the object. (Figure 4.1)
- The same map may be shared by many different locations, not necessarily restricted to locations of the same type. (For example, Figure 4.3)

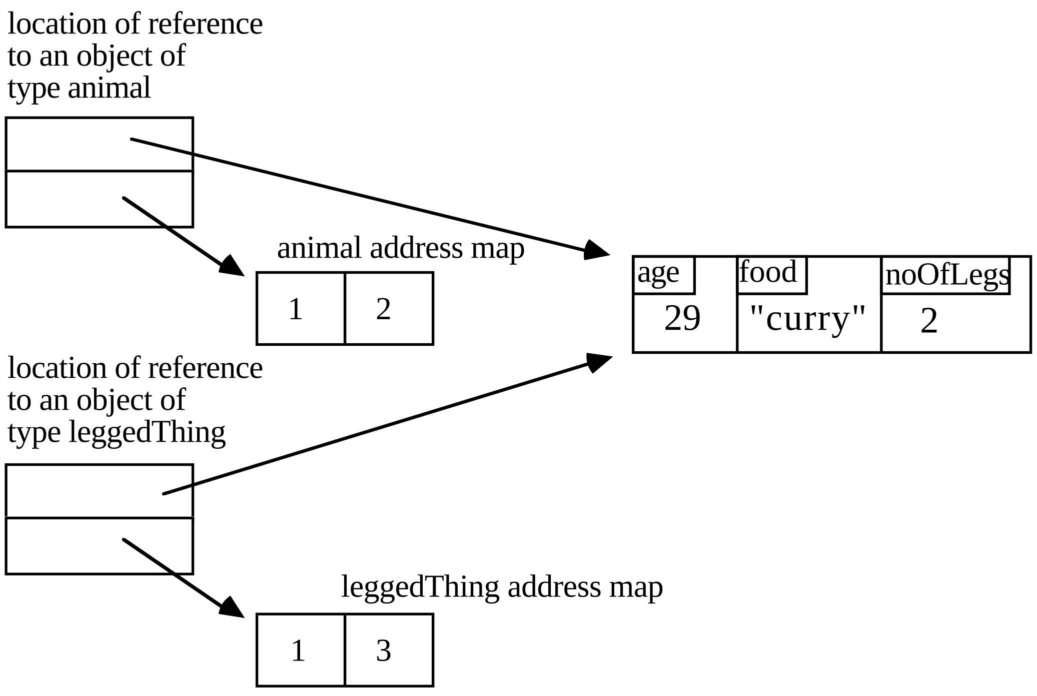

Figure 4.1 Location Address Maps

Use of the address map is as follows. The compiler statically calculates the field offset as if it were dealing with an object of the known supertype. If it knows that the object is of precisely this type, then this value is used to index the object directly. Otherwise, the calculated offset is used instead to index into the local address map for the object, which will contain the correct address of the required field. The penalty for a dereference is thus at most a single indirection.

When records are assigned, it is normally necessary only to perform a straightforward assignment of both the pointer to the record and the pointer to the current address map. This is not expected to incur any penalty on most machine architectures, which already support double word assignment.

More work is required only when the assignment involves the loss of type information, as happens when a subclass is assigned to a location of a superclass type. Where this occurs in a program is statically detectable. When it does happen, a new address map is constructed. This must map the indexes calculated for the supertype into the values contained in the appropriate positions in the subtype's address map, and may be constructed by performing the first level indirection for each field accessible by the supertype. The code to construct this mapping may then be generated statically.

Let us consider the following example:

```
let a = proc( lThing : leggedThing ; lAnimal : leggedAnimal )
begin
        .
        .
        lThing := lAnimal
        .
        .
end
```

Here the location *lThing* of type *leggedThing* has had an object of type *leggedAnimal* assigned to it; this is legal as far as the type rules are concerned, since *leggedAnimal* is a subtype of *leggedThing*. After the assignment, the location *lThing* must have an address map which correctly maps the fields *age* and *noOfLegs* to the appropriate addresses in the new object. Notice that in general it is not known statically that the object referenced by *lAnimal* does not have other fields which are not accessed from the location *lAnimal,* since it could itself be any supertype of *leggedAnimal.*

Where no optimisation is possible, as in this case, the new address map for *lThing* must be created dynamically. The size of the map is known, as it only need contain addresses for the fields which may be accessed from the new location, in this case *age* and *noOfLegs*. The address map objects however must be created dynamically, as a new one is required every time the piece of code is executed.

Code is planted by the compiler to first construct a new address map object of the required size. Then, for each field in the type being assigned to, in this case *age* and *noOfLegs*, two offsets are calculated by the compiler: let us call them X, the calculated offset into an object of the type being assigned to, and Y, the calculated offset into an object of the type being assigned. Code is then planted to copy the contents of the Yth location in the old address map into the Xth location in the new one. When this has been done for all the addressable fields, the new address map is complete. An example of this is shown in Figure 4.2.

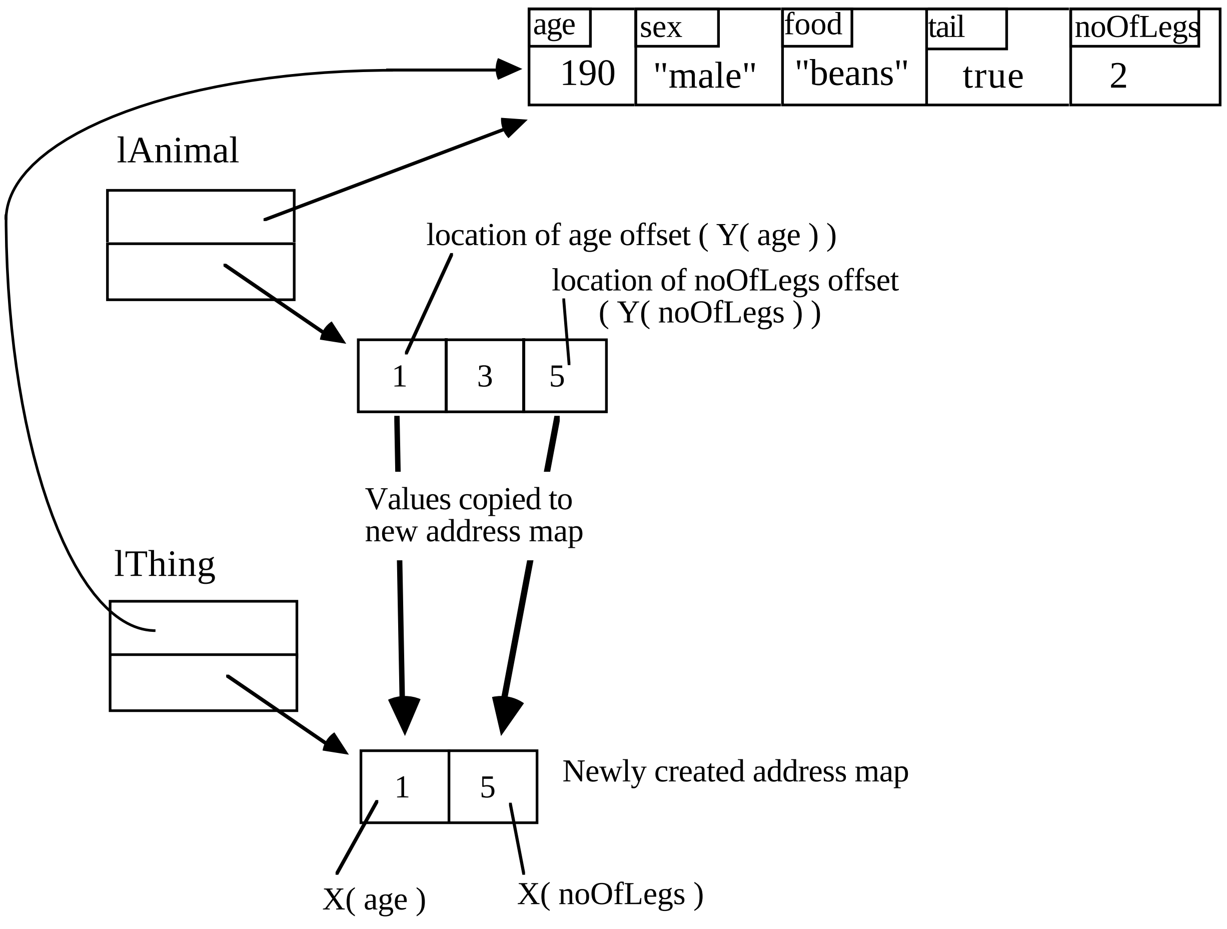
age
190
sex
"male"
food
"beans"
tail
true
noOfLegs
2
lAnimal
location of age offset (Y(age))
location of noOfLegs offset (Y(noOfLegs))
1
3
5
Values copied to new address map
lThing
1
5
Newly created address map
X(age)
X(noOfLegs)

Figure 4.2 The Mechanics of Assignment

If the precise type of the assigned value is known statically, then this process may be factored out and the address map may be constructed by the compiler rather than during execution. Notice that as these maps are immutable, they are required at most once per static assignment, rather than dynamically, and also that they may be shared between objects of different types.

The mechanisms described above for creating and calling bounded universally quantified procedures remain the same. As values within these procedures may still be assigned to supertypes, all addressing must be done using the same indirection techniques. The same technique of compiling 'wrapping' procedures is still required to act as temporary storage from which to build new address maps for the quantified parameters: as type widening is occurring, they cannot be simply assigned.

An optimisation of this technique may be obtained by realising that when a record is originally created, the address map required is normally a simple isomorphism: that is the *i*th address will contain the value *i*. There need therefore be only one of these maps for the entire system, so long as it is at least the length of the largest record. On initialisation, every object location shares this single isomorphic map. Assignment and construction of new address maps may then take place in the manner described.

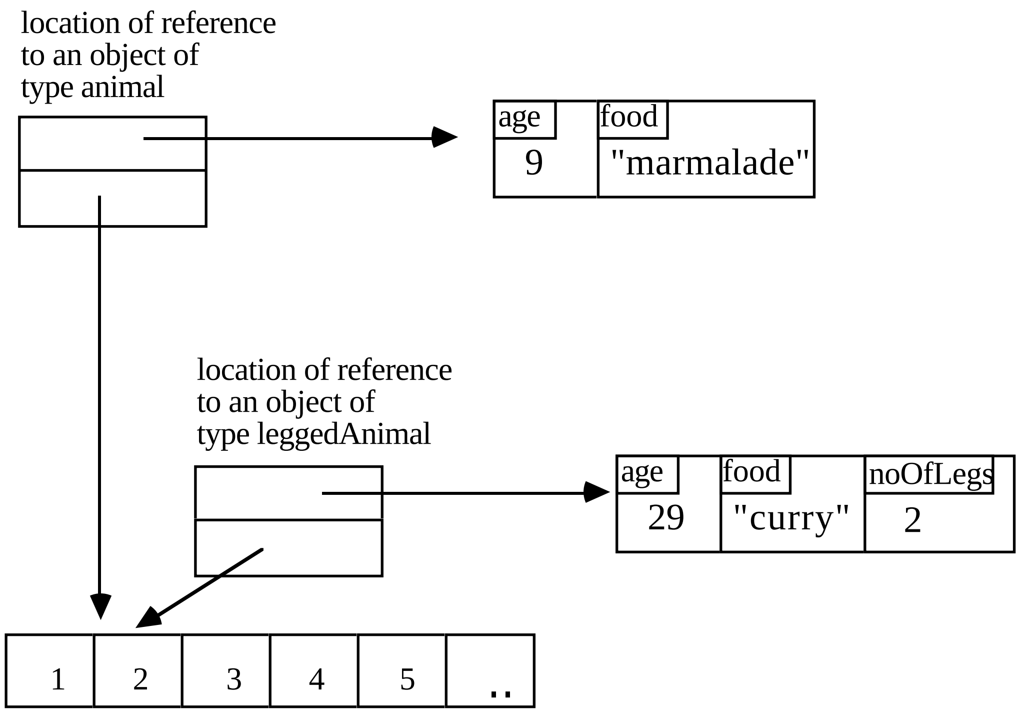
location of reference to an object of type animal
age
food
9
"marmalade"
location of reference to an object of type leggedAnimal
age
food
noOfLegs
29
"curry"
2
1
2
3
4
5
..

Figure 4.3 A Single Shared Address Map for Object Creation

This optimisation is most important to the efficient working of the scheme, as it means that an object creation never involves the creation of a new address map. Notice also that this is not a constraint for a distributed system: although only one of these maps is necessary, any number may be used to suit the implementation of the system.

Another important optimisation is when, on assignment, the fields of the supertype object are identical to those at the start of the subtype object. If this is the case, the original address map may be assigned, as it will still work correctly. This will always be true in languages with explicit single inheritance schemes.

5 Comparisons

In this section we will consider the various merits of the schemes discussed in this paper, namely the 'traditional' string symbol table address maps scheme and our addressing mechanism. When we examine the differences between these mechanisms we must consider the costs in four areas: space overhead, assignment, object creation, and indexing.

In both schemes the space needed to store a reference to the address map is the same – namely the space required by one pointer. However, using the 'traditional' mechanism the pointer is associated with the object instances. In our scheme the pointer to the addressing information is associated with the locations at which object references are stored. We assert that in most cases the number of object instances and the number of locations storing object references will be of the same order of magnitude.

However, the 'traditional' address map scheme requires the names of the fields to be stored in the address map. This may involve the construction of a simple table with low space overhead or an elaborate hash table. Using our scheme the names are no longer necessary with consequent space savings.

A very much smaller number of address maps is required in a system using our scheme. In particular, objects which are not assigned to a location occupied by a supertype never need have additional address maps created for them.

Therefore space is gained in our scheme due to a more compact address mapping scheme and the ability not to manufacture address maps in the general case. There may be some space lost due to extra references associated with objects.

The traditional object address map scheme has no additional cost associated with assignment. The time cost of the scheme described in this paper depends on the kind of assignment being performed. Usually assignments will be of an object of some type being assigned to a location of exactly the same type. If this is the case, the only additional cost is of an extra word assignment. The assignment of two contiguous words is not expected to be significantly more costly than the assignment of a single word. When a subtype is assigned to a location of a supertype, the operation depends on whether the type of the object to be assigned is known statically or not. If it is, then the new address map required may be created statically and the extra dynamic cost is a single pointer update. If it is not, then the map must be built dynamically, and the cost is one dereference and integer assignment for each field in the supertype.

In both schemes discussed the additional cost associated with object creation is low. Using the 'traditional' technique the address map would normally be created at the time of establishment of the type or class of the object. In our scheme the address map objects are not required upon creation, as a single system-wide map may be used until such time as a subtype assignment occurs.

The new addressing mechanism described in this paper has been optimised for indexing performance. By indexing we mean indexing objects to retrieve values, assigning values to locations within objects and method selection in 'traditional' object-oriented languages. In the worst case we have a single level of indirection for an index, and in the best case we have a statically planted offset.

The traditional address mapping mechanism uses a hash table to look up names in the address map. Even using a very efficient hashing mechanism such as that used by the Eiffel language implementation [mey85], the speed of an indirect address can never be equaled. Even if a very high hash hit rate is achieved there is still an associated cost with performing the hashing function in addition to the index. The technique described also allows for non-uniform field sizes and for the reorganisation of field order.

All of the examples given in this paper describe what we call first order multiple inheritance, and the records shown contain only simple objects. It should be noted that this was done for ease of description only, and that the technique is sufficiently general to work for any order of multiple inheritance with implicit subtyping.

6 Conclusion

We have discussed an addressing mechanism which we believe in many ways to be superior to the traditional methods used to implement addressing in object oriented languages. Whether this scheme is more efficient than the traditional methods depends on how the system under consideration is used. We have therefore shown where the trade-offs lie and leave the implementor to decide whether this scheme is suitable for the language or system under consideration.

7 Acknowledgements

We would like to acknowledge discussions on this matter with Stanley B. Zdonik of Brown University, and Tony Davie and Dave McNally of St Andrews University. We would also like to thank Jim Coplien of Bell Laboratories for his comments on the first version of this paper.

8 References

[ABC83] Atkinson M.P. Bailey P.J., Chisholm K.J., Cockshott W.P. & Morrison R. "An Approach to Persistent Programming". The Computer Journal, Vol 26, No 4 (December 1983) pp 360-365.

[AM85] Atkinson M.P & Morrison R. "Procedures as Persistent Data Objects", ACM ToPLaS, Vol 7, No 4 (October 1985) pp 539-559.

[BBB88] Bancilhon F., Barbedette G., Benzaken V., Delobel C., Gamerman S., Lecluse C., Pfeffer P., Richard P. & Valez F. "The Design and Implementation of O_2, an Object Oriented Database System". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. In *Lecture Notes in Computer Science, 334.* Springer-Verlag (September 1988) pp. 1-22.

[BCC88] Brown A.L., Connor R.C.H., Carrick R., Dearle A. & Morrison R. "The Persistent Abstract Machine Version 4.0". Universities of St Andrews and Glasgow PPRR-59 (1988).

[car84] Cardelli L. "A Semantics of Multiple Inheritance", *In Semantics of Data Types, Lecture Notes in Computer Science 173*, Springer-Verlag (1984) pp 51-67.

[CBC89] Connor R.C.H., Brown, A.L., Carrick R., Dearle A. & Morrison R. "The Persistent Abstract Machine". Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia. (January 1989) pp 80-95.

[CL88] Connors T. & Lyngbaek P. "Providing Uniform Access to Heterogenous Information Bases". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany. *In Lecture Notes in Computer Science, 334.* Springer-Verlag, pp. 162-173 (September 1988).

[GR83] Goldberg A. & Robson D. *Smalltalk-80: The Language and its Implementation,* Addison Wesley (1983).

[mey85] Meyer B. *Object-Oriented Software Construction,* Prentice Hall (1988).

[str86] Stroustrup B. *The C++ Programming Language,* Addison Wesley (1986).

[str87] Stroustrup B. "Multiple Inheritance for C++", EUUG - European Unix Systems User Group Newsletter, Volume 7 (1987).

[SZ86] Skarra A. & Zdonik S.B. "An Object Server for an Object-Oriented Database System", Proc. International Workshop on Object-Oriented Database Systems, Pacific Grove California (September 1986) pp 196-204.

[WZ88] Wegner P. & Zdonik S.B. "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like", *In Proceedings ECOOP '88 – European conference on Object-Oriented programming, Lecture Notes in Computer Science 322,* Oslo, Norway, (August 1988) pp 55-77.