Type Extension and Efficient AST Manipulation

K John Gough and Diane Corney*

Abstract

Oberon-2 is an object-oriented language with a class structure based on type extension. The runtime structure of Oberon-2 is described and the low-level mechanisms for dynamic type checking explained. It is shown that the superior type-safety of the language, when used for programming styles based on heterogeneous, pointer-linked data structures, has an entirely negligible cost in runtime performance.

1 Introduction

Various authors [1, 2] have described the features they deem necessary for a language to be classed as *object-oriented*. There are many different points of view possible. However we take the minimalist view that the essential features which make a language fit this general description are —

(a) some kind of class structure, with inheritance

(b) some means for dynamically binding methods to objects based on runtime tests of object class

Oberon-2 [3, 4] is a minimal language which provides these features. Oberon-2 is an extension of Oberon [5] the main feature of which is type extension [6]. Oberon, in turn, was based on Modula-2 [7].

Object oriented languages of this kind seem to us to be natural vehicles for the implementation of software systems based on abstract syntax tree (AST) representations. We are therefore interested in reasoning about the efficiency of the code produced by implementations of such languages for these kinds of programs.

*Both authors are with the Faculty of Information Technology, QUT, Brisbane

In this paper we describe the key features of Oberon-2 as they relate to the implementation of runtime type tests. We then describe the way in which such tests are used in programs which manipulate ASTs, and describe the translation of these key features. Finally, we give our conclusions.

2 The Oberon-2 Type System

2.1 Type Extension

Oberon-2 provides for a class hierarchy based on record structures. When a record type is declared, other record types can be declared which are extensions of the first. For example —

```
T0 = RECORD
       x,y : INTEGER;
     END;

T1 = RECORD (T0)
       z : REAL;
     END;
```

The record type *T0* contains two fields, x and y. The record type *T1* contains three fields, x, y and z. *T0* is called a **base type**, and *T1* is an **extension** of *T0*. We denote this $T1 \triangleright T0$. *T1* could then be further extended. Note that types may be extended only by the addition of fields.

This mechanism provides for tree-like hierarchies of types to be constructed, thus allowing for a kind of *single inheritance*. Hierarchies defined for record types automatically extend to pointers to these types.

```
T0Ptr = POINTER TO T0;
T1Ptr = POINTER TO T1;
```

In this example, *T1Ptr* is an extension of *T0Ptr*. Again we denote this $T1Ptr \triangleright T0Ptr$. We often

need to express the idea that one type is a (possibly trivial) direct or indirect extension of some other type. We denote this reflexive transitive closure of the *extends* relation by the symbol $\rhd^*$. The negative of this predicate is denoted $\not\rhd^*$.

The semantics of assignment of record types allows for extensions to be assigned to base types. Such an assignment has projection semantics, that is, only those fields belonging to the base type are copied from the right hand side expression. For example, suppose we have the following variable[1] declarations —

```
t0    : T0;
t1    : T1;
t0ptr : T0Ptr;
t1ptr : T1Ptr;
```

Then the following predicates hold —

`t0 := t1;` is valid, since *T1* $\rhd^*$ *T0*
`t1 := t0;` is invalid, since *T0* $\not\rhd^*$ *T1*

Similarly for pointers to records —

`t0ptr := t1ptr;` is valid, since *T1Ptr* $\rhd^*$ *T0Ptr*
`t1ptr := t0ptr;` is invalid, since *T0Ptr* $\not\rhd^*$ *T1Ptr*

Projection semantics also apply to the substitution of actual parameters for formal parameters of value mode in procedure calls. In the case of formal parameters of variable mode, an actual parameter may be of any type which is in the $\rhd^*$ relation with the formal type. In this case the called procedure sees the whole of the record, and may query its actual type.

In the case of the assignment of **pointer objects**, an assignment may change the type of the designated object. In this case, as in the case of variable mode formals, the expression must be in the $\rhd^*$ relation with the assignment target.

Oberon-2 is a statically typed language in which every object has a static type which can be determined at compile time, even in the presence of separate compilation. **Reference objects,** that is, objects of pointer types and variable mode formal parameters, have a dynamic type which must be determined at runtime. The static semantics of the language ensure that the dynamic type T_d and the static type T_s are always related by the predicate $T_d \rhd^* T_s$.

Statically or automatically allocated records are not reference objects and in this case the dynamic type is always equal to the static type, that is $T_d = T_s$. The same is true of value mode formal parameters.

The bound types of pointer objects are dynamically allocated records and have a dynamic type such that $Tb_d \rhd^* Tb_s$, as for the pointer types themselves.

Visibility of objects is controlled at the level of granularity of record fields. Each field of a record type may be exported, not exported or exported in *read only mode*. Access control is thus elegant, and assists in establishing program invariants.

The range of statements and type structuring mechanisms is similar, in general terms, to the Pascal family of languages. However, as will be shown, the existence of the type extension mechanism makes it possible to replace Pascal's inelegant and confusing union construct (the so-called variant "record") with a safe and elegant mechanism.

2.2 Dynamic Binding

Procedures can be declared as applying to a particular record type. Such procedures are called **type-bound procedures**. An example of such a declaration might have a header —

```
PROCEDURE (VAR node : T) Walk (a,b : X);
```

In this example the variable *node* of the bound type T is called the **receiver**. The parameter list on the right has the usual semantics. This procedure is bound to the record type T. The syntax allows for the receiver (called *self* in many languages) to be given a meaningful name.

Extended types automatically inherit the type-bound procedures of ancestor types. However, an extended type may have new type-bound procedures declared for it, or may have procedures which override or extend the behaviour of the

[1] For all the examples in this paper, types have names which begin with an upper case letter. Variables have names which begin with lower case. This is our local coding convention, but is not part of the language.

corresponding procedures bound to the ancestor types. The original type-bound procedure and the one which overrides it must share the same signature.

When a program calls a procedure with such an overloaded name the actual procedure which is invoked is determined from the dynamic type of the object. This may require runtime selection.

2.3 Type Testing

Apart from the dynamic binding of procedures there are two syntactic mechanisms which refer to dynamic type. These are the **type guard** and the **type test**.

Type Guards

A type guard asserts that some used occurrence of an identifier designates an entire variable which at runtime will have the specified type, or some extension of that type. The assertion allows the compiler to safely generate code to access all the fields defined for that sub-hierarchy of the type structure. These accesses need no type tests at runtime, apart from the executable assertion. Failure of any type guard results in program abortion.

The syntax is —

designator	→	*ident* { selector }.
selector	→	'.'*ident* | '[' exp ']' | '↑' | typeGuard.
typeGuard	→	'(' *typeIdent* ')'.

A guard $t(T)$ asserts that *DynTyp(t)* $\triangleright^*$ T.

With our example declarations, a typical type-guarded designator might be —

```
t0ptr(T1Ptr)^.z
```

This example asserts that at runtime the variable *t0ptr* has the dynamic type *T1Ptr* (or an extension of *T1Ptr*). Thus, access to the field z is valid.

If the type guard were not present then a compile time error would result, as a field z is not known for the record type *T0*, which is the static type to which *t0ptr* is bound.

The *regional* type guard has the same semantics as the single type guard, but applies to all used occurrences of the specified identifier within a statement sequence. A regional type guard allows successive sub-hierarchies to be tested sequentially. For example —

```
WITH t0ptr : T1Ptr DO
  < statements1 >
| t0ptr : T0Ptr DO
  < statements2 >
END;
```

Because the type guard tests if the dynamic type of the variable is the specified type or an extension of it, the ordering of the cases within this construct is meaningful. If the test for *T0Ptr* were placed first, then that option would always be executed, as *t0ptr* is statically of type *T0Ptr* and at runtime can only be *T0Ptr* or an extension. The regional guard may also contain an *ELSE* clause which is executed if none of the other type guard tests is true. If no test is successful and there is no *ELSE* clause then the program aborts.

Type Tests

The type test is a predicate which evaluates the same test as a type guard. The Boolean result depends on the success or failure of the type test.

The syntax is —

typeTest	→	*ident* 'IS' typeNam.

The Boolean expression t `IS` T is a runtime evaluation which returns the predicate *DynTyp(t)* $\triangleright^*$ T. A typical use, with our example declarations is —

```
t0ptr IS T1Ptr
```

2.4 Runtime Type Descriptors

Because a reference object may have dynamic type which differs from the static type, there must be some mechanism to find information about the dynamic type of an object at runtime. Every record object must thus contain a reference to a runtime type descriptor which denotes the type of the object. Figure 1 is a diagramatic representation of a typical structure. All objects of the same dynamic type reference the same descriptor.

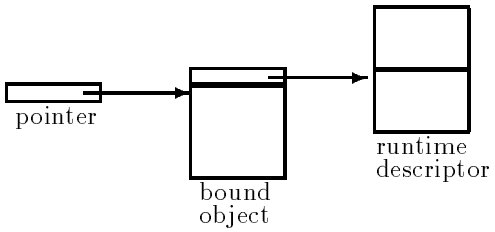

Figure 1: Relationship of pointer, bound object and runtime descriptor

In this figure, the runtime descriptor consists of two parts. There is an area which contains pointers to the type-bound procedures. The offset of every type-bound procedure within every runtime descriptor is known at compile time, and may thus be called by a constant-time indexing operation.

The second area contains an array of pointers to the descriptors of all the types of which this type is an ($\triangleright^*$) extension. The top element always points to itself since every type is a trivial extension of itself. This array may be thought of as analogous to the *display vector* used for access to non-local data in some implementations of *Algol*-like languages. This concept was introduced in a recent paper by Cohen [8]. A suprising result of this organisation is that the predicate $Tx \triangleright^* Ty$ may be evaluated in constant time. This has spectacular consequences for the efficiency of these tests.

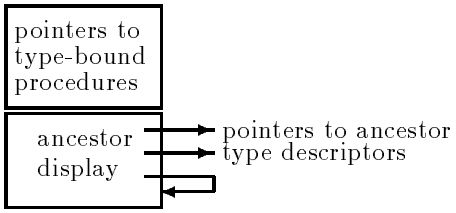

Figure 2: Details of descriptor structure

Every type descriptor contains two type-bound procedures which are used by the internal workings of the system. One is an *initialiser thunk*, and the other is a garbage collector *helper thunk*.[2] Figure 2 has more detail of the descriptor structure.

[2]Although we do not deal with these facilities here, Oberon-2 supports safe and efficient automatic garbage collection. In this area it is in stark contrast to (say) C++.

3 Abstract Syntax Trees and Oberon-2

A very flexible way of constructing software tools depends on the realisation of abstract syntax trees (ASTs) by pointer linked data structures [9]. Various formalisations have been introduced to describe such structures [10]. In conventional languages such heterogeneous structures are implemented using variant records/union types. This is necessary, of course, because in strongly typed languages a pointer is only bound to a single type.

In such structures *tag fields* serve to distinguish the components of the union, and allow a program to select manipulative actions depending on the particular tag value. Unfortunately, the cost of performing runtime tests on the values of tags is prohibitive, and almost no compilers support this. This follows from the fact that at each level of field-selection in a designator a test for set membership needs to be performed at runtime. The lack of such runtime checking is a major insecurity in the implementation of these styles of programs.

As noted above, Oberon-2 permits an alternative approach for creating and manipulating such structures. This approach is based on dynamic binding of procedures and conditional control flow predicated on type tests. There is a widely held misapprehension that this approach is inefficient.

We are currently implementing compilers for Oberon-2. The prototype implementations are written in Modula-2 and have front-ends which are based on abstract syntax trees. These are similar in philosophy to the gardens point modula-2 compilers [11], and share common backends with native code versions of **gpm**[12]. An obvious question which occurred to us was to ask if the manipulations of the AST in the front-end would be more or less efficient if translated into Oberon-2[3].

[3]The ETH compilers for Oberon-2 are written in Oberon-2, but do not use the type extension facilities in the way which we suggest. Instead they use structures which are direct products of their notional variants. This is, of course, just as insecure as the untested variant tags of the Pascal family.

4 Type-mediated Accesses in the AST

The following section will deal with examples of the various ways of accessing nodes in an *AST*. These examples will be illustrated using Modula-2, and presume the use of variant records with a tag field indicating the type of the variant. The tag is assumed to be some enumerated type. To demonstrate the efficiency of such a method, the high level code is then translated into the corresponding assembly language instructions. The examples shown here use *MIPS* assembly code, but the conclusions are similar for other load/store architectures. The code may be understood with reference to Figures 1 and 2. After all the code is shown for the Modula-2 access method, the Oberon-2 alternative is shown, and the same steps followed. The resulting assembly code for the two alternatives may then be compared.

4.1 Case Statement

The first exemplary situation is the traversal of the nodes of a tree structure. The *AST* used here is a tree of identifier descriptors. The *IdDesc* types are pointers to records. The tree is traversed using a case statement to determine what action is performed at each node. In this instance, the tag field of the record must be accessed to determine which case is executed. This means a procedure of the following form would be used to traverse the nodes —

```
PROCEDURE TraverseId (VAR id : IdDesc);
BEGIN
  CASE id^.idClass OF (* tag value *)
  | varCls   : ......
  | constCls : ......
  | procCls  : ......
  ...
  END;
END TraverseId;
```

Notice that we have a single procedure, the effect of which is selected, inside the procedure, based on the actual tag value. For case statements, the address of the target code is calculated using a jump table. The general form of the jump table is shown in Figure 3.

Each case has a label, referenced in the jump table, and ends with an unconditional branch to the exit label.

```
        .data
jmptbl: .word label1
        .word label2
        ...
        .word labelN
```

Figure 3: Layout of the jump table

The assembly language instructions which would subsequently be produced for this scenario are —

```
lw   r1,id      ; load id value to r1
lbu  r2,8(r1)   ; byte tag at offset 8
bgtu r2,max,deflbl ; default if too big
mul  r3,r2,4    ; scale index by wordsize
la   r4,jmptbl  ; load jmptbl address
add  r5,r4,r3   ; r5 <- base + offset
lw   r6,(r5)    ; fetch target address
j    r6         ; jump to it
```

In Oberon-2 this single procedure would be replaced by a set of procedures, one bound to each extension of some base *IdRecord* type. The code in each of these *Traverse* procedures would be equivalent to the code for the corresponding case in the above case statement. Thus instead of a single procedure call with various cases selected *after* the call we have a set of procedures, with the selection occurring *during* the procedure call.

To traverse an identifier record in Oberon-2, the statement *id.Traverse;* calls whichever procedure is bound to the runtime type of *id*. The assembly instructions are —

```
lw  r1,id      ; load id value to r1
lw  r2,-4(r1)  ; descriptor ptr at -4
lw  r3,24(r2)  ; proc at offset 24 say
jal r3         ; call proc in reg
               ; jal is subroutine call
```

As can be seen, the code sequence for calling a type-bound procedure is actually shorter than that for the case statement. In each case a single procedure call/return overhead is involved, but the type-mediated selection of the procedure *prior* to the call is faster than the indexed jump in the case statement *after* the call.

4.2 Explicit Type Tests

Another access idiom in union-based *AST*s, occurs with an explicit test for a particular tag value. In Modula-2 this would be of the form —

```
IF id^.idClass = procCls THEN ... END;
```

The resulting assembly instructions are —

```
lw   r1,id    ; load id value to r1
lbu  r2,8(r1) ; tag at offset 8 say
bne  r2,7,label ; ORD(procCls) == 7
              ; branch if <> to lit
```

The equivalent control flow in Oberon-2 requires a type test. The statement required would be —

```
IF id IS ProcTyp THEN ... END;
```

where *ProcTyp* is the "proc" extension of the pointer to the base record type.

In assembler this becomes —

```
lw  r1,id     ; load id value to r1
lw  r2,-4(r1) ; descriptor ptr at -4
lw  r3,8(r2)  ; assume ancestor offset
              ; is 8 in the display
la  r4,ProcPtr_Desc ; load adr
bne  r4,r3,label
              ; branch if not equal
```

In this case there is a slight overhead for the runtime test. Note however that in the case of nested variants (corresponding to multilevel extensions) there would be multiple tag tests, but a single, constant time type test suffices. Recall that the type test does not test for equality, but for membership in a subtree of the class hierarchy.

4.3 Direct Access to a Record Field

The third idiom we wish to consider is the direct access of a field of the record type. In the case of the variant record the access might be —

```
id^.varInfo
```

This access assumes that at runtime the *active variant* will be one for which the specified field exists. As indicated earlier, it is so costly to check for this in general that compilers leave correctness for the programmer to ensure. The (unchecked) access to such a field is very simple. Here is the assembly code —

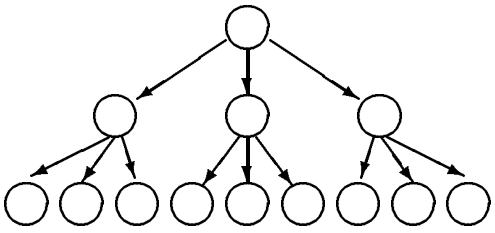

Figure 4: An example type hierarchy

```
lw r1,id     ; load id value
lw r2,12(r1) ; field offset == 12
```

In Oberon-2 in order to access a field which exists only in some sub-class of some base type requires a type guard if the variable has been statically declared as being of the base type. The code is —

```
id(VarPtr)^.varInfo
```

In this case we assert that the dynamic type of *id* will be a (possibly trivial) extension of the *VarPtr* type.

Assembly code generated would be —

```
lw  r1,id     ; load id value to r1
lw  r2,-4(r1) ; descriptor ptr at -4
lw  r3,8(r2)  ; index ancestor display
la  r4,VarPtr_Desc ; load address
bne r3,r4,trap ; abort if <>
lw  r2,12(r1) ; else get the field at
              ; known offset 12 say
```

In effect, the price to be paid for the complete safety of this method is an additional four instructions for this access.

In order to see why runtime tag tests are difficult, consider the situation where logically a union type has the hierarchical structure of Figure 4. Such a structure may be realised in Modula using either nested variants with two levels of tags, or as a "flat" union of the 9 leaf types.

If the flat structure is adopted, access to a first level field requires a set membership test on the value of the single tag field. The multilevel approach requires only equality tests on the tags, but requires two tests to validate access to a second level field.

These tests are very costly in general, given the lack of semantic constraint on the tag types which are acceptable to the Pascal-like languages. With

disciplined design, relatively low cost explicit tests may be inserted by the programmer. In the flat case, **gpm**'s nonstandard inline *Assert* facility might be used as follows —

```
Assert(id^.idClass IN okSet);
... id^.varInfo ...
```

Even recognising the common subexpression in this example leaves an overhead of five extra instructions with best coding.

Note that the situation with multiple tags is no better. In principal a single tag equality test could guarantee membership of a complete type subtree. However, access to the lower level tags must be protected by tests on the higher level tags. It is this mixing of data and type information which lies at the heart of the inefficiency.

Finally, we note that when records are assigned, under some circumstances an implicit test of the dynamic type is required. This does not happen except when explicit pointer dereference is used to obtain copying rather than aliasing semantics. The cost of these tests appears to be similar to that of the range checks on assignments which are a familiar part of the implementation of Pascal-like languages. In many cases, the use of the regional guard allows even this cost to be amortized over several accesses.

5 Conclusions

In this paper we have considered several cases where runtime testing of the dynamic type of objects is required. The code for carrying out these tests and the code for the equivalent actions in a procedural language have been compared.

Contrary to widely held belief, the overheads for the runtime testing appears to be negligible. Much of this reduction of overhead is caused by the elegant mechanism for type resolution in constant time. In return, at least for these styles of software tools, important enhancements of type safety have been obtained.

Because we are using common code generator back-ends for our Modula-2 and Oberon-2 compilers we expect to be able to quantify these matters precisely in terms of time taken for the execution of realistic programs. We shall report these results in due course.

References

[1] Meyer, B., *Object-oriented Software Construction*, Prentice Hall, 1988.

[2] Booch, G., *Object-oriented Design with Applications*, Benjamin/Cummings Publishing, 1991.

[3] Mössenböck, H. and Wirth, N., "The Programming Language Oberon-2", Computer Science Report 160, ETH Zurich, May 1991

[4] Mössenböck, H. "Object-Oriented Programming in Oberon-2", Computer Science Report, ETH, Zurich.

[5] Wirth, N., "The Programming Language Oberon", *Software Practice and Experience*, Vol 18, No 7, pp 671-690, July 1988.

[6] Wirth, N., "Type Extension", *ACM Trans. on Prog. Lang. and Systems*, Vol 10, No 2, pp 204-214, April 1988.

[7] Wirth, N., "Programming In Modula-2", Editions 1-4, Springer Verlag, 1982, 1983,1985,1988.

[8] Cohen, N.H., "Type-Extension Type Tests Can Be Performed in Constant Time", (Technical Correspondence), *ACM Trans. on Prog. Lang. and Systems*, Vol 13, No 4, pp626-629, October 1991.

[9] Gough, K.J., *Syntax Analysis and Software Tools*, Addison-Wesley, 1988.

[10] Evans, A., Jr., and Butler, K.J., Eds, "DIANA Reference Manual", Revision 3, Tartan Laboratories, Pittsburg, PA, 1983.

[11] Gough, K.J., Cifuentes, C., Corney, D., Hynd, J., Kolb, P., "An Experiment in Mixed Compilation/Interpretation", *Australian Computer Science Communications*, Vol 14, No 1, January 1992.

[12] Gough, K.J., "The D-Code Compiler Front-end GP2D", QUT Report, FIT 1/92, 1992.