Lightweight Parametric Polymorphism for Oberon

Paul Roe and Clemens Szyperski

Queensland University of Technology, Brisbane QLD 4001, Australia

Abstract. Strongly typed polymorphism is necessary for expressing safe reusable code. Two orthogonal forms of polymorphism exist: inclusion and parametric, the Oberon language only supports the former. We describe a simple extension to Oberon to support parametric polymorphism. The extension is in keeping with the Oberon language: it is simple and has an explicit cost. In the paper we motivate the need for parametric polymorphism and describe an implementation in terms of translating extended Oberon to standard Oberon.

1 Introduction

A key goal of Software Engineering is to support the production and use of reusable code. Reusable code, by definition, is "generic" i.e. applicable in a number of different contexts. To guarantee that code is reused correctly strong typing is desirable. Genericity in code can best be expressed by polymorphic types. Two different forms of polymorphism have been identified: inclusion and parametric [2]. In theory inclusion and parametric polymorphism are orthogonal concepts and neither can be used to satisfactorily replace the other.

The Oberon language supports inclusion polymorphism via subtyping. This permits a certain degree of safe code reuse to be achieved. However parametric polymorphism cannot be safely expressed. This paper describes how parametric polymorphism can be incorporated into Oberon. The following results are achieved:

- strict extension to Oberon
- relatively simple extension in the style of Oberon
- orthogonal to subtyping
- polymorphic routines can be statically type checked
- requires no change to a compiler back-end or run-time system
- polymorphic code is compiled to real object code shared by all instantiations
- polymorphic libraries can be separately compiled e.g. for DLLs
- incurs no run-time overhead, and can eliminate some type tests

The rest of this paper is organised as follows. Section 2 briefly describes Oberon and its subtyping, for further information see e.g. [10, 12]. Section 3 describes the extension to Oberon. The implementation of parametric polymorphism is described in Section 4. The final sections describe further work, related work and conclusions.

2 Motivation: Inclusion Polymorphism and Oberon

In Oberon we may define a list of heterogeneous objects (records) thus:

```
TYPE Base      = POINTER TO BaseDesc;
     BaseDesc  = RECORD END;

     List      = POINTER TO ListDesc;
     ListDesc  = RECORD elem: Base; next: List END;
```

The `Base` type is the type of list objects. For the list to support different sized objects, objects must be represented via a reference. By extending the `Base` type we can support lists of different objects, for example:

```
     Point     = POINTER TO PointDesc;
     PointDesc = RECORD (BaseDesc) x,y: REAL END;

     Emp       = POINTER TO EmpDesc;
     EmpDesc   = RECORD (BaseDesc)
                     name, address: String;
                     eno, level: INTEGER
                 END;
```

The Oberon syntax `RECORD (BaseDesc) ...END` indicates that the record type extends an existing record type (namely `BaseDesc`). It is also possible to further extend the `EmpDesc` and `PointDesc` objects.

To process a heterogeneous list of objects we need to distinguish between different objects in order to determine how to process them. This is achieved by dynamic type inspection or dynamic dispatch. Different procedures (methods), but all with the same interface, may be bound to different types; thus depending on the type, different procedures may be invoked. For example we may bind a print method to each of the objects in the previous example, thus:

```
PROCEDURE (r: Base) Print;
END Print;

PROCEDURE (r: Emp) Print;
BEGIN Out.String(r^.name)
END Print;

PROCEDURE (r: Point) Print;
BEGIN Out.Real(r^.x,0); Out.Real(r^.y,0)
END Print;
```

In the examples above `r` is the receiver object, a.k.a. *self*. (Note, strictly the dereferencing operation (`^`) used above is redundant, but we leave it for those readers more familiar with Pascal or Modula-2.) An object of type `Base`, `Emp` or `Point` can be printed thus: `x.Print`. A general list printing procedure can be written as follows:

```
PROCEDURE ListPrint(l: List);
   VAR t: List;
BEGIN
   t := l;
   WHILE t # NIL DO
      IF t^.elem # NIL THEN t^.elem.Print END;
      t := t^.next
   END
END ListPrint;
```

This procedure is polymorphic, it will work on lists of any objects having type at least `Base`. Furthermore it may be reused with new types of objects unknown at the time `ListPrint` was written. If a new type is created which is a subtype of `Base`, lists can be formed containing such objects, and they can be printed using the above procedure: without any re-compilation or modification of `ListPrint`. This kind of reuse arises from inclusion polymorphism (subtyping).

We may also define other useful routines such as a list length function:

```
PROCEDURE ListLen(l: List): INTEGER;
   VAR len: INTEGER; t: List;
BEGIN
   t := l; len := 0;
   WHILE t#NIL DO INC(len); t := t^.next END;
   RETURN len
END ListLen;
```

and a list prepend procedure:

```
PROCEDURE ListPrepend(VAR x: List; y: Base);
   VAR z: List;
BEGIN
   NEW(z); z^.elem := y; z^.next := x;
   x := z
END ListPrepend;
```

This list prepend procedure will prepend any object of type `Base` to any `List`. Thus it always allows a heterogeneous list to be constructed. However sometimes a more constrained version of prepend is required. For example an application may utilise a list of employee descriptions (`Emp`), and it may be that the list should only contain `Emp` or its subtypes. Thus we would like to prevent `Base` type, `Point` type or any other type of element not a subtype of `Emp` from being prepended to the list. This constraint should be statically enforceable by the type system. However this is not possible in Oberon, all we can do is define a new list type to contain `Emp` elements, and new functions on the type: thereby eliminating code reuse – our goal.

Another example of a routine which cannot be adequately expressed in Oberon is the list map operation. This operation invokes a procedure on all elements of a list:

```
TYPE BaseProc = PROCEDURE (x: Base);

PROCEDURE ListMap(p: BaseProc; l: List);
   VAR t: List;
BEGIN
   t := l;
   WHILE t#NIL DO
      p(t^.elem); t := t^.next
   END
END ListMap;
```

Unfortunately the procedure argument must have an argument of type `Base`; this is necessary since the list may be heterogeneous. However if we have a list of at least `Point` type elements, and we wish to rotate them, we *cannot* use the following procedure:

```
PROCEDURE Rotate(r: Point);
   VAR t: REAL;
BEGIN
   t := r^.x; r^.x := -(r^.y); r^.y := t
END Rotate;
```

Instead we must use a type test to dynamically check that the type really is at least `Point`:

```
PROCEDURE BRotate(r: Base);
BEGIN Rotate(r(Point))
END BRotate;
```

(The type guard `r(Point)` asserts that `r` has at least type `Point`, if not the program aborts.) This is a poor solution for implementing homogeneous lists. We have no static guarantee that a homogeneous list will be constructed, and even if it is we must pay the cost of unnecessary type tests and extra procedure calls. Effectively what the above example has done is to simulate parametric polymorphism in Oberon. However in doing so we are programming in an untyped way, with no static checking, rather like programming in Smalltalk. For example there is nothing to stop a program being compiled with an invocation like `ListMap(BRotate,l)` where `l` is a list of `Emp`. The following section describes an extension to Oberon which supports type checked parametric polymorphism.

3 Parametric Polymorphism for Oberon

We introduce parametric polymorphism via our previous example. A type may be parametrised on types in much the same way as a procedure may be parametrised on values. For example our previous list example may be parametrised on an element type thus:

```
TYPE Ptr(A)      = POINTER TO A;
     List(A)     = Ptr(ListDesc(A));
     ListDesc(A) = RECORD elem: Ptr(A); next: List(A) END;
```

We term `List` and `ListDesc` polymorphic (or poly) types and similarly we term a type with no type parameters a monomorphic (or mono) type. Actual type parameters of poly types may be "forward-declared". We shall use the poly type `Ptr` in place of `POINTER TO` in the remainder of this paper. In general a type may have multiple type parameters. Some example lists are declared below:

```
VAR emplist       : List(EmpDesc);
    pointlist     : List(PointDesc);
    listpointlist : List(ListDesc(PointDesc));
```

A list having type `List(EmpDesc)` is statically guaranteed by the type system to only contain elements of type at least `EmpDesc`. In addition to parametrising types we can also parametrise procedures on types, for example:

```
PROCEDURE <A> ListPrepend (VAR x: List(A); y: Ptr(A));
   VAR z: List(A);
BEGIN
   NEW(z); z^.elem := y; z^.next := x;
   x := z;
END ListPrepend;
```

Procedure type parameters are listed before the procedure name, enclosed in angle brackets. The reason for this is that, unlike poly types, procedures do not have actual type parameters supplied to them when they are used (invoked). Procedure type parameters are scoped over the whole of the procedure body. The type parameter in `ListPrepend` enables the procedure to operate on different types of lists; furthermore the prepended element is guaranteed (by the type system) to have the same, or a more specific, type as the list. For example the polymorphic list prepend procedure can be invoked thus: `ListPrepend(emplist,e)`. Although no actual type parameters are supplied each invocation is statically checked to ensure the element being prepended matches the type of list elements.

The previous map function may be rewritten:

```
TYPE Proc(A) = PROCEDURE (x: Ptr(A));

PROCEDURE <A> ListMap (p: Proc(A); l: List(A));
   VAR t: List(A);
BEGIN
   t := l;
   WHILE t#NIL DO
      p(t^.elem); t := t^.next
   END
END ListMap;
```

For example, this may be invoked thus: `ListMap(Rotate,pointlist)` or `ListMap(IncLevel,emplist)`, where `IncLevel` is defined thus:

```
PROCEDURE IncLevel(e: Emp);
BEGIN INC(e.level)
END IncLevel;
```

By combining Oberon's inclusion polymorphism (subtyping) with parametric polymorphism, the map procedure can be abstracted to work for entire families of collections, not only lists:

```
TYPE Collection(A) = Ptr(CollectionDesc(A));
     CollectionDesc(A) = RECORD
        elem: Ptr(A)
     END;

     List(A) = Ptr(ListDesc(A));
     ListDesc(A) = RECORD (CollectionDesc(A))
        next: List(A)
     END;

     Tree(A) = Ptr(TreeDesc(A));
     TreeDesc(A) = RECORD (CollectionDesc(A))
        left, right: Tree(A)
     END;

PROCEDURE <A> (c: Collection(A)) Map (p: Proc(A));
BEGIN p(c.elem)
END Map;

PROCEDURE <A> (l: List(A)) Map (p: Proc(A));
BEGIN
   l.Map^(p);  (* "super call" to overridden procedure *)
   IF l.next # NIL THEN l.next.Map(p) END
END Map;

PROCEDURE <A> (t: Tree(A)) Map (p: Proc(A));
BEGIN
   IF t.left # NIL THEN t.left.Map(p) END;
   l.Map^(p);
   IF t.right # NIL THEN t.right.Map(p) END
END Map;
```

For example, it is now possible to abstractly map `Rotate` over a collection of points `Collection(PointDesc)`, regardless of whether the collection is actually a list, a tree, or any other structure derived from the base type `Collection(A)`.

3.1 Language Extension

This section describes details of the parametric polymorphism extension to Oberon. The syntax of Oberon [10] is extended, by modifying TypeDec, ProcDec and Type thus:

```
TypeDec    = IdentDef [TypePars] "=" Type
TypePars   = "(" [Ident {"," Ident}] ")"
ProcDec    = PROCEDURE [PTypePars] [Receiver] IdentDef [FormalPars] etc.
PTypePars  = "<" [Ident {"," Ident}] ">"
Type       = Qualident [TypeActs] | etc.
TypeActs   = "(" [Type {"," Type}] ")"
```

Our language extension is a minimal one; a particular goal was to make explicit any costs associated with parametric polymorphism, in keeping with Oberon. A polymorphic value, by its very name, may have any shape and size. Some languages support arbitrary use of polymorphism. However, supporting this either entails an implicit run-time cost or prevents compilation of polymorphic routines. This is discussed further in Section 5.

Our solution to the problem is a simple one. We only support references to poly values, analogous to subtyping restrictions in Oberon. That is all poly values must be: explicitly referenced via pointers to poly values, implicitly referenced via `VAR` parameters or procedure types. Thus all costs of polymorphism are explicit.

Our approach could be broadened to also support polymorphism over any pointer sized objects. This trick is used by C programmers and for supporting Modula-2 opaque types. However this is rather implementation dependent, and may necessitate the introduction of procedure wrappers and closures, hence we prefer not to do this.

The following describes our language extension. Two new forms of type abstraction (type parametrisation) are introduced:

1. Parametrised type declarations, e.g. `TYPE List(A) =` T_{exp}`;`. The type parameter `A` is scoped over the single type declaration T_{exp}.
2. Parametrised procedure declarations, e.g. `PROCEDURE <A> Foo (l: List(A))`, but not parametrised procedure type declarations[1]. For example a procedure type may be declared thus: `P(A) = PROCEDURE (l: List(A))`, but the following is *illegal:* `P = PROCEDURE <A> (l: List(A))`.

The following restrictions on the use of poly types and type parameters guarantee the explicit cost model previously described. A poly type or type parameter can only be used in the following type contexts:

1. a pointer to a type parameter, e.g. `POINTER TO A`
2. a var parameter, e.g. `PROCEDURE <A> Foo (VAR x:A)`
3. a poly type application, where the actual type parameter is:

[1] This restricts our system to rank one polymorphism.

   (a) a record type, e.g. `List(EmpDesc)`
   (b) a type parameter, e.g. `List(B)`
4. an uninstantiated polymorphic receiver object[2], for example: `PROCEDURE <A> (List(A)) Foo` is legal, but `PROCEDURE <A> (List(EmpDesc)) Foo` is *illegal*.

Instantiated poly record types do not have subtype relationships induced by their actual type parameters; this follows naturally from standard Oberon, and is strictly necessary to be type safe. For example although `PointDesc` is a subtype of `BaseDesc`, `List(PointDesc)` is *not* a subtype of `List(BaseDesc)`. This is the reason why methods cannot be bound to instantiated poly types.

A variable of poly type has no accessible methods or fields, unless its type is constrained via a type guard or `WITH` statement. The type guard, type test and `WITH` constructs can all operate on polymorphic values, e.g:

```
PROCEDURE <A> Foo (p:Ptr(A));
BEGIN
   IF (p#NIL) & (p IS Emp) THEN p(Emp)^.eno := 123 END
END Foo;
```

A procedure may introduce new type parameters. Type parameters are scoped over the whole procedure definition. Angle brackets are used to distinguish type parameters from procedure receiver objects, and also to indicate that they do not need to be, and indeed cannot be, supplied on procedure invocation. This latter point is important, we do not want to overburden the program or programmer with specifying types; actual types of actual parameters are supplied implicitly to procedures, so are poly types. We wish to encourage the reuse of parametric poly procedures; thus syntactically they should be no more expensive to use than ordinary monomorphic procedures.

3.2 Type Checking Rules

The instantiation of a poly type does not create a new type; therefore poly types are type equivalent by structure. Type aliasing of poly types follows that of ordinary Oberon types.

Within a procedure declaration, a type parameter is only type compatible with itself; thus the following is type correct:

```
PROCEDURE <A> Foo (x: Ptr(A)): Ptr(A);
BEGIN RETURN x
END Foo;
```

but the procedure below is type *incorrect*:

[2] In addition the most general receiver objects `PROCEDURE <A> (r:Ptr(A)) Foo` and `PROCEDURE <A> (VAR r:A) Foo` are also illegal.

```
PROCEDURE <A,B> Foo (x: Ptr(A)): Ptr(B);
BEGIN RETURN x
END Foo;
```

since `x` has type `Ptr(A)` which is not type compatible with `Ptr(B)`. The procedure header states that `A` and `B` may be different types; therefore they are not type compatible. Effectively in the procedure, `A` and `B` are abstract types which cannot be manipulated other than by: inspecting their type with type tests and type guards, and passing them to other polymorphic procedures.

To understand how a procedure invocation is typed it is necessary to explain what type a poly procedure has. Polymorphic procedures may be used with different types, therefore we can represent a poly procedure's type as a universally quantified type. For example consider the three procedure headers below:

```
PROCEDURE <A> Id(x: Ptr(A)): Ptr(A);
PROCEDURE <A,B> K(x: Ptr(A); y: Ptr(B)): Ptr(A);
PROCEDURE <A> Pick(x,y: Ptr(A)): Ptr(A);
```

We may give them the following formal types:

$\forall A.$ (`Ptr(`A`)): Ptr(`A`)`
$\forall A, B.$ (`Ptr(`A`), Ptr(`B`)): Ptr(`A`)`
$\forall A.$ (`Ptr(`A`), Ptr(`A`)): Ptr(`A`)`

To type the use of a procedure (call or assignment to a procedure variable) its type must unify with the types of its context and any actual parameters. Consider the program fragment below:

```
TYPE S = Ptr(Srec); Srec = RECORD END;
     T = Ptr(Trec); Trec = RECORD (Srec) END;
     U = Ptr(Urec); Urec = RECORD (Srec) END;
VAR s:S; t:T; u:U;
```

An expression such as `Id(t)` will type with the unifier $A \mapsto$ `T`, and hence `t:=Id(t)` or `s:=Id(t)` will type correctly. Similarly `K(t,u)` will type with unifier $A \mapsto$ `T`, $B \mapsto$ `U`. However in general subtyping requires a more sophisticated form of unification to be used. For example the expression `s:=Pick(t,s)` should type, but the expression `t:=Pick(t,s)` should not (a valid implementation of `Pick` is to simply return its second argument). To handle these cases unifiers need to be refined using a least upper bound rule for subtypes. For example consider the expression `Pick(t,s)`, matching the type of `t` to A, in the type of `Pick` above, should produce a unifier $A \mapsto$ `T`. Subsequently matching `s` to A with the previous unifier $A \mapsto$ `T` should produce a refined unifier (least upper bound of `T` and `S`) of $A \mapsto$ `S`. Thus the result of `Pick(t,s)` has type `S`, not `T`. Similarly the result of `Pick(t,u)` has type `S`; since the least upper bound of `T` and `U` is `S`.

4 Implementation

In this section we describe an implementation of the parametric polymorphism extension to Oberon. The implementation of the extended type system is briefly

described and code generation is described in terms of rewrite rules, which map extended Oberon to standard Oberon. The rewrite rules show how our extension can be implemented, that the extension has no associated run-time cost, and that no modifications to the run-time system or compiler back-end are necessary. A prototype system to type check extended Oberon and translate it to standard Oberon is being constructed [17].

4.1 Type Checking

Type checking can be performed by a modified Oberon type checker which performs unification and refinement of unified subtypes. It is desirable to reduce all poly types to normal form. This aids type checking by eliminating any unnecessary intermediate poly types. The body of a poly procedure can be simply checked by treating all poly types as new types. Poly procedures can be used with different types. Therefore when a poly routine is typed, its type parameters must be distinguished from those of different uses of the same routine. Furthermore when a context constrains an instance of a type parameter all other related instances must also be constrained. To implement this when a poly procedure is used its formal type is copied and fresh type variables are substituted for bound type parameters. This is a standard technique, see e.g. [1]. Type variables are instantiated, like logical variables, during type checking, and are also subject to refinement, as described in the Section 3.2.

4.2 Rewrite Rules

Our rewrite rules map extended Oberon to standard Oberon. Poly values (instantiated or not) are restricted to be references; these fall into three categories: explicit record pointers, implicit record pointers via var parameters or function pointers in the case of poly procedure values. Poly values in each category have the same representation for different record types; this is already required in order for standard Oberon's subtyping to work. Thus, there is no difference between the run-time representation of ordinary values in each category from poly ones; since all values in each category have the same representation. (Note that not all mono values can be used in poly contexts.) Therefore all that is required is to map an extended Oberon program to correct standard Oberon. We assume that the program has already been type checked by the extended Oberon type checker which checks poly and mono types.

We can identify two places where the rewrite rules must map extended Oberon to standard Oberon: type declarations and type expressions, and program statements and expressions. In the first case all poly type declarations of the form: `TYPE T(`$A_1 \ldots A_n$`) = TE` are rewritten to `TYPE T = TE'`, where `TE'` is the result of recursively rewriting the type expression `TE`. Any instance of a type parameter in a type expression is rewritten to `ANYRECORD` the standard base record type[3]. Similarly any procedure declaration introducing poly types is

[3] We assume a standard type `ANYRECORD` that is the base type of all records that have

rewritten without them, recursively mapping any instances of type parameters to `ANYRECORD`.

The second case concerns program statements and expressions. As previously mentioned all poly types, instantiated or uninstantiated, have the same representation as their mono type instances. Thus to coerce values between poly types and mono types (and vice versa) requires no change of representation; all that is required is a cast between the appropriate types. This can be achieved using `SYSTEM.VAL`; a caveat is that the exact behaviour of this operation is undefined[4]. The following rewrites are necessary:

1. Any dereference or `NEW` operations on instantiated poly pointer types require the pointer to be cast to the appropriate pointer type.
2. A procedure object may be used in a more specific or more general context than its type would allow; in which case it must be cast to the type required by its context.

Note a type guard on a poly type effectively instantiates the poly type.

An example rewrite is shown below; the extended Oberon program:

```
TYPE Ptr(A)      = POINTER TO A;
     List(A)     = Ptr(ListDesc(A));
     ListDesc(A) = RECORD elem: Ptr(A); next: List(A) END;
     Proc(A)     = PROCEDURE (x: Ptr(A));

VAR el: List(EmpDesc); pl: List(PointDesc);

PROCEDURE <A> ListMap (p: Proc(A); l: List(A));
   VAR t: List(A);
BEGIN
   t := l;
   WHILE t#NIL DO
      p(t^.elem); t := t^.next
   END
END ListMap;
...
NEW(el); NEW(el^.elem);
```

is rewritten to:

```
TYPE Ptr      = POINTER TO ANYRECORD;
     List     = Ptr;
     ListDesc = RECORD elem: Ptr; next: List END;
     Proc     = PROCEDURE (x: Ptr);
```

---

no declared base type. If unavailable, such a type can be introduced as part of the rewriting process.

[4] We utilise `SYSTEM.VAL` in a completely general way. For some compilers we would need to translate some casts a little differently.

```
VAR el: List; pl: List;

PROCEDURE ListMap (p: Proc; l: List);
   VAR t: List;
BEGIN
   t := l;
   WHILE t#NIL DO
      p(SYSTEM.VAL(ListDesc,t^).elem);
      t := SYSTEM.VAL(ListDesc,t^).next
   END
END ListMap;
...
NEW(SYSTEM.VAL(POINTER TO ListDesc,el));
NEW(SYSTEM.VAL(Emp,(SYSTEM.VAL(POINTER TO ListDesc,el))^.elem));
```

5 Related Work

Many languages have facilities for supporting some sort of parametric polymorphism. Table 1 categorises some prominent examples by the properties of their respective facilities.

language group	generics are type checked	coexistence with inclusion polymorphism	real object code generated from generics	all run-time costs are explicit
CLU, Ada 83	yes	none	no	yes
ML, Napier-88	yes	none	yes	no
C++, Modula-3	no	orthogonal	no	yes
Ada 95	yes	orthogonal	no	no
Eiffel, Sather, Theta	yes	bounded poly	yes	no
Extended Oberon	yes	orthogonal	yes	yes

Table 1. Categories of Parametric Polymorphism

Approaches that neither support type checking of polymorphic code, nor generation of shared object code really are just glorified macros (Modula-3 generic modules, C++ templates).

Constrained parametric polymorphism was first introduced for CLU [6, 7]. Parametric polymorphism for a Pascal like language was proposed by Tennent in [16]. All functional programming languages in the tradition of ML, and some related languages, such as Napier-88 [9] support parametric polymorphism. Many modern object-oriented languages also support parametric polymorphism, including Eiffel [8], Sather [14, 15], or Theta [3, 5]. In all these languages the run time cost of using parametric polymorphism is hidden.

Hidden costs are caused by the use of uniform representations for all values in shared polymorphic code. Such representations have to cater for all types, irrespective of size. Usually the uniform representation is an indirection to the natural representation: often called a *boxed representation*. When a value is moved between a poly and mono context, coercion is used to convert values between representations: often called boxing/unboxing. However boxing and unboxing is an implicit cost involving hidden operations such as heap allocation. Furthermore boxing can necessitate the use of closures to support partial applications of procedures to types. In extended Oberon we avoid these hidden costs by restricting poly types to references.

In untyped languages such as Lisp or Smalltalk, where every context is polymorphic, a uniform representation must be used everywhere, e.g. Lisp *S-expressions*. However this is rather inefficient. Optimisation, such as dynamic compilation, is possible, but still incurs hidden costs.

A way to avoid hidden costs without restricting instantiations is to give up on sharing code and instead generate code for each instantiation, e.g. Ada style generics. This leads to code explosion and prevents construction of generic dynamically linkable libraries. In addition to leading to code explosion, Ada-style generics are relatively heavyweight, as they require explicit instantiation before use and therefore make extensions to existing generics subtle and complicated[5].

Various unsafe practices that do not cause any run-time cost and allow for the sharing of object code are in common use to simulate parametric polymorphism in languages that do not support it, demonstrating the *need* for safe and efficient language-level support. Examples are unsafe type casts in C or unsafe constructs closed off by multiple safe interfaces such as Gough's Device (Chapter 7 in [4]).

Recently some work similar to ours has investigated parametric polymorphism for an abstract imperative language called Polymorphic C [13]. This also restricts parametric polymorphism to pointers. However, the emphasis of this work is more theoretical than ours: a formal type system is presented and type soundness proven, but no implementation exists. Our work concentrates on the smooth practical integration of parametric polymorphism with an existing language, Oberon. The relationship between our work and Polymorphic C requires further investigation.

6 Further Work

There are several areas which we wish to pursue further. It is desirable to support parametric polymorphism over arrays. The present implementation strategy could support this providing references to arrays (explicit and implicit) have the same representation as references to records; since a poly procedure must be capable of accepting either references to records or arrays. An alternative would be to perform a run-time test, but this is not in keeping with our zero cost implementation strategy.

[5] Extensions have to define a local instantiation that is parametrised with the type parameters of the extending generic.

More general combinations of inclusion and parametric polymorphism are possible, for example bounded polymorphism [2]. With the polymorphism introduced effectively all poly types must be at least `ANYRECORD`, for example:

```
PROCEDURE <A> ListMap (p: Proc(A); l: List(A));
```

the `ListMap` procedure will operate on all values of type `A`, from `ANYRECORD` downwards in the subtype hierarchy. The procedure can assume nothing about the type of `A`, and hence its fields and methods. Thus it is rather like saying `A` must be at least of type `ANYRECORD`. A more powerful form of polymorphism than that proposed here allows the type of `A` to be bounded. However, there are also known pitfalls of general bounded polymorphism [11].

It is desirable to develop a library of poly routines and poly types for Oberon, as has been successfully done for Smalltalk and C++ (STL).

We have produced a translator which rewrites extended Oberon to standard Oberon; we wish to incorporate extended Oberon into our standard compiler.

7 Conclusions

Parametric polymorphism supports type parametrisation, just as procedures support code parametrisation. As such, this is a proven and well-established concept and part of many languages. Parametric polymorphism does not interfere with or substitute inclusion polymorphism (subtyping), but the two mutually benefit from each other. When properly designed and implemented, parametric polymorphism improves type safety, maintainability and reusability of code.

In current languages, parametric polymorphism comes at one of two costs. Either polymorphic code cannot be separately compiled into shared generic object code, or the use of such shared code incurs significant hidden costs. Languages in the former category simply generate code for each instantiation of a generic, leading to code explosion and, possibly worse, preventing the construction of generic dynamic link libraries. Languages in the latter category, which hide run-time costs, can significantly affect the programmer's control over execution cost in time and space.

In this paper we presented a simple extension to the language Oberon that is lightweight in all dimensions. The extension is carefully restricted to simultaneously support type checking and full separate compilation of polymorphic code *and* to not introduce any run-time cost in time or space. The extension is not only simple but also strictly upwards compatible with Oberon. The extension only affects the front-end of a compiler: after type checking, extended Oberon can be rewritten into standard Oberon.

Acknowledgements

We would like to thank Jürgen Wendel for implementing our ideas in the form of an extended Oberon to Oberon translator, and for his comments on a draft of this paper. This work was partially funded by ARC grant ARCSG 55, 7056.

References

1. A V Aho, R Sethi, and J D Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1986.
2. L Cardelli and P Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
3. M Day, R Gruber, B Liskov, and A Myers. Subtypes vs. Where Clauses: Constraining parametric polymorphism. In *Proc, 10th Conf on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 156–168, October 1995.
4. K J Gough and G M Mohay. *Modula-2: A Second Course in Programming*. Prentice Hall, 1988.
5. B Liskov, D Curtis, M Day, S Ghemawat, R Gruber, P Johnson, and A C Myers. Theta Reference Manual, preliminary version. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1995.
6. B Liskov and J Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
7. B Liskov, A Snyder, R Atkinson, and C Schaffert. Abstraction mechanisms in clu. *Comm ACM*, 20(8):564–576, August 1977.
8. B Meyer. *Eiffel – The Language*. Prentice Hall, 2 edition, 1992.
9. R Morrison, A Dearle, R C H Connor, and A L Brown. An ad-hoc approach to the implementation of polymorphism. *ACM TOPLAS*, 13(3):342–371, 1991.
10. H Mössenböck. *Object-Oriented Programming in Oberon-2*. Springer Verlag, 1993.
11. B C Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1):131–165, July 1994.
12. M Reiser and N Wirth. *Programming in Oberon – Steps beyond Pascal and Modula*. Addison-Wesley, 1992.
13. G Smith and D Volpano. Towards an ML-style Polymorphic Type System for C. In *1996 Eurpoean Symposium on Programming*, Linköping, Sweden, April 1996.
14. D Stoutamire and S Omohundro. Sather 1.1. Technical report, International Computer Science Institute, Berkeley, CA, 1996.
15. C Szyperski, S Omohundro, and S Murer. Engineering a Programming Languager— the Type and Class System of Sather. In *Proc, 1st Intl Conf on Programming Languages and System Architectures*, number 782 in Springer LNCS, Zurich, Switzerland, March 1994.
16. R D Tennent. *Principles of Programming Languages*. Prentice Hall Int., 1981.
17. J Wendel. Parametric Polymorphism for Oberon. Technical report, Faculty of Information Technology, QUT, Brisbane, in preparation.

This article was processed using the LaTeX macro package with LLNCS style