Polymorphism in a Compiled Language

G.V. Cormack and A.K. Wright

CS-86-27

Polymorphism in a Compiled Language

G. V. Cormack and A. K. Wright

Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

ABSTRACT

A prototype compiler for the language **ForceOne** is described. **ForceOne** gives the programmer the ability to define formal parameters whose corresponding actual parameters may be of arbitrary type. In addition, identifiers and operators may be overloaded, with the appropriate definition being selected using available type information. Finally, routines may be defined by the user to convert automatically between types. Each of these language features facilitates *polymorphic* programming. **ForceOne** differs from competing languages in that it has *all* of these facilities and in that it is truly compiled – it does not rely on the ability to enumerate all possible types during compilation; it does not require a separate "generic" parameterization mechanism and compilation pass; it does not require run-time name and type resolution. Here we present a brief description of **ForceOne**, and of the techniques used in compiling it.

1. Introduction

A type system in a programming language is a set of rules that determine what values each symbol may represent, and how these values may be manipulated. In a statically typed language, we attempt to assert at compile time that these rules are not violated. Static typing offers a number of advantages in execution efficiency and security: we believe it is the essence of what is commonly known as a compilable language. However, traditional static typing exacts a large price because of its lack of flexibility. This lack of flexibility is evident when the programmer frequently has to re-express essentially the same program fragment because the type system prevents him from abstracting it.

Polymorphism is the ability to define an abstraction that transcends traditional static typing [cf. Mil 78]. For example, the FORTRAN symbol + is polymorphic in that it applies equally well to `INTEGER`s and `REAL`s. In this paper we describe three language facilities that enable the user to define his own polymorphism, namely: overloading, automatic type conversion, and parameters of arbitrary type. These facilities are found in the statically typed language **ForceOne**, for which a prototype compiler has been written. While each of these facilities exists in various forms in existing languages, we feel that the particular approach described here is a novel combination of simplicity, expressive power, and straightforward implementation.

The following section describes briefly the programming language **ForceOne**. The next section addresses implementation considerations in compiling **ForceOne**. The final sections contrast the design and implementation of **ForceOne** with related efforts, and offer some closing remarks.

2. The language ForceOne

This section describes the features of **ForceOne** that are relevant to polymorphic programming. While **ForceOne** is a new language, the features described here could be added cleanly to a number of type secure languages like ISO Pascal [Pas 80]. Features of **ForceOne** that are not germane to polymorphism are not described. In particular, the facility for modular structure and information hiding is not described: this facility is completely *separate* from the type structure, in contrast to many other modern languages. A more complete description of **ForceOne** is found in [Wri 86], and a rationale appears in [Cor 86]. **ForceOne** is a successor to the languages **L** [Cor 81, Cor 83, Lec 84] and **M** [Jud 85], and is a sibling of the language **Zephyr** [Cor 85].

Some very simple examples are presented to illustrate the basic features of the language. We hope that these are sufficient to whet the reader's imagination to discover interesting applications. A grammar describing the relevant parts of **ForceOne** appears in the appendix.

2.1. Type overview

Each object in **ForceOne** has a type which describes the set of values it may take on. The functions which operate on objects of a particular type are not considered to be part of that type. Types are not first-class values; they are manipulated only at compile time.

A type declaration introduces a new type which is implemented as some base type, but is nevertheless unique. The type declaration also defines three functions, `:=`, `retype` and `detype`. `:=` is the assignment operator; `retype` and `detype` convert from the base type to the new type and from the new type to the base type respectively. These functions are overloaded (see section 2.7). By restricting access to `retype` and `detype`, the programmer can make the new type opaque. By providing automatic conversions (see section 2.8), the programmer can make the new type essentially interchangeable with the base type.

2.2. Basic types and declarations

Primitive types in **ForceOne** include `int`, `real`, `bool`, `char`, and `void`. `ref` types describe storage that holds a value of a particular type, e.g. `ref int`. `proc` types describe executable procedures that yield a particular type as a result, e.g. `proc bool`. `record` types contain fields of various types; declaration of a `record` type also defines functions to access these fields.

Declarations have the form

identifier : *type* == *expression*

This declaration binds to *identifier* the value yielded by *expression*. The value associated with *identifier* does not change throughout its lifetime. However, if *type* is a `ref` type, the storage referenced by *identifier* may be assigned various values. In this instance, *expression* is typically a call to the stack storage allocator `new` (in fact, if == *expression* is omitted, == `new` is assumed).

In place of *type* we may use the keyword `type`; in this case *expression* must be some type `t`, and *identifier* names a new type whose base type is `t`. For example,

```
intvar: type == ref int
```

2.3. Parameterization

A typical parameterized procedure is declared thus:

```
scale: [ i: real, scalefactor: real ] proc real ==
    { return scalefactor * i }
```

Here, `scale` is of type

```
[ real, real ] proc real
```

An application of `scale` has the form

```
    scale [ actual parameter, actual parameter ]
```

where *actual parameters* are expressions of type `real`. The same notation is used for arrays.

2.4. Implicit parameters

In the example above, `i` and `scalefactor` are both explicit parameters to `scale`. In the following declaration, `scalefactor` is made an implicit parameter:

```
    scale: [ i: real | scalefactor: real ] proc real ==
        { return scalefactor * i }
```

As far as the definition of `scale` is concerned, implicit parameters differ from explicit parameters only in that they are separated by `|`. However, the caller does not specify an actual parameter to correspond to `scalefactor`; there must exist in the calling scope a definition of the name, and it is passed automatically as the actual parameter. Implicit parameters allow the caller to create a local environment that affects the way the procedure behaves.

2.5. Parameters of arbitrary type

It is not necessary to specify completely the type of a formal parameter. All or part of the formal type may be replaced by

```
    ? identifier
```

where *identifier* is a user-specified name. For example,

```
    cube: [ x: ?t ] proc t
```

declares a procedure `cube` that accepts an argument of arbitrary type `t` and returns a value of type `t`. `t` is in fact a parameter to cube (`t` is of type `type`), and may be used anywhere a type is required (for example, it is used as the result type). `t` is called a *query* parameter. Like implicit parameters, query parameters are not specified explicitly at the call site, but are used like any other parameters within the definition of the parameterized procedure.

Like all types, type parameters carry with them no operations. In order to define the body of `cube`, we require the operation * on values of type `t`. This operation is typically supplied as an implicit parameter to `cube`:

```
    cube: [ x: ?t | '*': [t, t] proc t ] proc t ==
        { return x * x * x }
```

Thus, `cube` provides an abstraction that will cube the value of its argument, provided that the operation * is available at the site where `cube` is called. Typical applications of `cube` might be:

```
    cube[2]        -- 8
    cube[3.0]      -- 9.0
```

`cube` would not normally accept parameters of type `string`, for example, because a * operation is not defined for strings. However, we could define such an operation, and be able to use `cube` with the scope of the declaration:

```
    {
        '*': [ a: string, b: string ] proc string ==
            { return concat[a,b] }

        cube[ "xyz" ]        -- "xyzxyzxyz"
    }
    cube[ "abc" ]            -- invalid; no * defined
```

We wish to contrast this approach with those that require adding * to the *definition* of the type `string`, to which we have no ready access.

2.6. Parameters of constrained type

We may constrain the set of possible actual types matched by a formal parameter. If we wish to match only references to storage, but the storage referenced may be of any type, we may specify the formal type

`ref ?`*identifier*

If we wish to match parameterless procedures, regardless of their result type, we specify the formal type

`proc ?`*identifier*

When an actual procedure is passed as a parameter, *identifier* will represent its result type. A formal parameter that matches procedures that have a parameter of arbitrary type is:

`[ ?`*identifier* `] proc void`

It is also possible for a formal parameter to match an actual parameter whose type contains a query parameter:

`[ ??`*identifier* `] proc void`

Here, the double `?` indicates that *identifier* belongs to the nested parameter list. In this example

```
fred: [ i: int ]  proc int  == ...
john: [ j: real ] proc real == ...
dick: [ k: ?t ]   proc t    == ...

poly1: [ z: [  ?q ] proc q] proc void == ...
poly2: [ w: [ ??r ] proc r] proc void == ...
```

`poly1` will accept `fred` or `john` as an actual parameter, while `poly2` will accept `dick`. In other words, the `??` notation allows us to pass polymorphic procedures as parameters to other procedures. `?q` is a parameter to `poly1` where `??r` is a parameter to `w`, which is in turn a parameter to `poly2`. In general, the number of `?` symbols determines the level of parameter nesting to which the query parameter belongs.

2.7. Overloading

When a user defines a new type, it is essential that he be able to extend existing notations to apply to this new type. For example, if we add the type `complex` to a language that does not have it already, we must be able to define the operations `+`, `-`, `/`, etc. for complex numbers. Furthermore, it is perfectly sensible to wish to define these same operations between the existing type `real` and the new type `complex`. Essentially, all the definitions of + form a common abstraction*.

In **ForceOne**, the same symbol may be overloaded to represent many values, provided each value has a distinct type. The programmer specifies that a name is to be overloaded by prefixing its declaration with the keyword `overload`. For example:

```
complex: type == ...
overload '+': [a: complex, b: complex] proc complex == ...
overload '+': [a: complex, b: real   ] proc complex == ...
overload '+': [a: real,    b: complex] proc complex == ... .
```

Only symbols so declared are overloaded; an overload declaration hides any previous non-overload declarations and a non-overload declaration hides all overload declarations of the same symbol.

When an overloaded symbol is used, the appropriate definition is selected to match the type required by context. This type information includes the number and type of any parameters, and the result type.†

* At the moment, we have nothing but good faith to ensure that all overloaded definitions of + contribute to the same abstraction. Ideally, there should be some formal specification of the meaning of + that all definitions should meet. Such a formal specification is far beyond the scope of this paper.

† These rules are similar to those of Ada [Ada 83]. They differ in that overloading must be specified explicitly, and

A number of other features of **ForceOne** rely heavily on overloading. The assignment operator := is merely an overloaded procedure, and may be defined by the user as he sees fit. Field selection functions for records merely overload other definitions with the same name – the type of the record ensures that the appropriate function is selected. Polymorphic parameters tacitly require overloading – the function `cube` (section 2.5) assumes that the appropriate overloaded definition is selected to match the implicit parameter `*`.

2.8. Automatic type conversion

Automatic type conversions are defined by overloading the special names `widen` and `narrow`. The intent is that widenings are conversions that involve no loss of information; narrowings may destroy information. If the symbols in an expression cannot be matched without conversion, widenings will be done to match actual parameters to formal parameters. If an expression cannot be matched with widenings alone, narrowings will be used. Widenings are done as early as possible in the evaluation of an expression (as deeply as possible in the expression tree) while narrowings are done as late as possible (as near the root as possible). Only one conversion will be inserted by the compiler at any one point in the expression tree.

In the definition of `complex` above, we can avoid having to declare three versions of `+, -, /,` etc. by providing a widening from `real` to `complex`:

```
overload '+': [a: complex, b: complex] proc complex == ...
overload widen: [a: real] proc complex == ...
```

In this example, providing one widening permits one definition of + (and `-, /,` etc.) to replace three. Let us consider a slightly larger situation. It is reasonable to have each of the types `int`, `real`, `longreal`, `complex`, and `longcomplex`, and to define all the binary arithmetic operations between any pair. Without automatic conversions, this would require 15 separate definitions per operator. With the introduction of 9 widenings this number is reduced to 5 (one per operation per type). With 4 widenings and 4 narrowings (to and from `longcomplex`,) a single definition is sufficient (if not efficient).

Automatic conversions also allow new types to be handled by existing routines. For example, stream input and output are done only on strings; complex numbers can be printed once a widening from `complex` to `string` is defined. Similarly, literals can be written as strings once a narrowing from `string` to `complex` is defined.

3. Implementation of ForceOne

To compile **ForceOne**, we apply mainly standard techniques used to compile block-structured languages. We rely heavily on the ability to have parametric procedures, and we require only the ability to allocate stack storage whose size is not known at compile time; heap storage allocation is not used. Algorithms for generating such code are fairly simple and are already well known.

3.1. Types and type parameters

Types in **ForceOne** have two roles: they determine the representation of a data object, and they control the selection of overloaded symbols. The first role is played both at compile time and at run time; the second role is purely a compile-time consideration. In this section we consider the first role.

The run-time value of a type is simply a representation specification (in the current implementation it consists of *size* and *alignment*). Whenever a new type is declared, these values can be computed easily from the base type or types. Parameters of type `type`, which form the basis for polymorphic procedures, are passed as a single number indicating both size and alignment. Therefore, the following procedure is implemented in the same way as a procedure with a single integer parameter and returning a pointer result:

---

that symbols of any type may be overloaded (not just procedures). The selection rules in **ForceOne** are further affected by the presence of query parameters and automatic conversions.

```
f: [t: type] proc ref t == ...
```

In order to modify `f` to return a result of type `t` instead of `ref t`, our run-time system is required to return a result whose length is determined by the parameter `t`:

```
f: [t: type] proc t == ...
```

Within `f` the parameter `t` is a new type distinct from all others. Therefore, the only things `f` can do with `t` are to use it to declare a local of type `t` or to pass it to another procedure. Thus procedures with a type parameter alone are useful in few applications.

In addition to a type parameter, functions that operate on values of that type are usually desired. The type parameter may be used in the specification of subsequent formal parameters. At the call site, when the type parameter is bound to an actual type, that actual type is substituted in the formal types of the other parameters. In the following example, `swap` can interchange the value of two storage locations, provided the values can be assigned.

```
swap: [ t: type, a: ref t, b: ref t, ':=': [ref t, t] proc void ]
        proc void ==
    {
        temp: ref t
        temp := a
        a := b
        b := temp
    }
```

A typical call to `swap` might be:

```
swap[int, x, y, ':=']
```

Let us consider first the header information describing `swap` that is stored in the symbol table. As for any procedure, the type of each formal parameter must be stored, as well as the type of the result. However, the types of parameters `a`, `b`, and `:=` refer to the previous parameter `t`. These dependencies are represented by special back pointers in the list describing the formal parameters. The result type may also contain such a back pointer. At the call site, the parameters are bound left-to-right, and therefore all types represented by back pointers have been bound to actual parameters before they are used. In the example, `t` is first bound to `int`, and then `a`, `b`, and `:=` are bound to `x`, `y`, and `:=` as if the formal parameter list were:

```
swap: [ t:type, a:int, b:int, ':=': [ref int, int] proc void ]
        proc void
```

Once this substitution is done, the actual parameters are passed in the normal way.

3.2. Implicit parameters and query parameters

Implicit parameters and query parameters are handled in much the same way as described above: the executable code is identical. The only difference is that the compiler automatically supplies the corresponding actual parameters. The natural way to write `swap` is:

```
swap: [ a: ref ?t, b: ref t | ':=': [ref t, t] proc void ]
        proc void == ...
```

The corresponding call to `swap` would be:

```
swap[x,y]
```

Here the specification

```
a: ref ?t
```

declares two formal parameters: `t` which is a query parameter of type `type`, and `a` which is of type `ref t`. Only one actual parameter, `x`, is bound to such a double formal parameter; its value is bound to `a` and part of its type (`int`) is bound to `t`. In the parameter type list for `swap`, `t` appears the same as any parameter of type `type`, but with an indication that there is no corresponding actual parameter. The type description for `a` has a back pointer to `t`, and an indication that this is a defining reference.

The implicit formal parameter := also appears in the formal type list much as before, except that it also is marked as having no explicit actual parameter. Instead, its symbolic name becomes part of the type entry. When it comes time to bind this parameter, the symbol table is searched (at the call site) to find an entry with the same name and type. This definition is then used as the corresponding actual parameter, and is passed in the normal way.

The roles of the three types of formal parameters can be summarized as follows: at the call site, the set of available definitions must be searched using name and type to find the object to be passed. With explicit parameters, the actual parameter contributes the name and the formal parameter contributes the type. With query parameters, the actual parameter contributes both the name and the type. With implicit parameters, the formal parameter contributes both the name and the type.

3.3. Nested query parameters

Query parameters prefixed by a single `?` are treated as described above whether or not they appear inside a nested parameter list. That is, they are parameters in the outer parameter list, and are filled in from the type of the actual parameter. A query parameter with more than one `?` serves merely to constrain the type of the actual parameter – it must have a query parameter in the same place that is prefixed by exactly one fewer `?`.

Consider the procedure `poly1` (section 2.6). `poly1` has two parameters `z` and `q` which are bound to the actual parameters `fred` and `int` or `john` and `real`. On the other hand, `poly2` has only one parameter, `w`, which is bound to `dick`. The query parameter `??r` indicates that `dick` must have a query parameter, (`?t`) and must return a result of the same type. `??r` and `?t` are bound together only while matching the type of `dick` with the type of the formal parameter `w`. `r` is not defined within the body of `poly2`.

3.4. Overload and type resolution

The algorithm that resolves types and selects the appropriate definitions of overloaded identifiers is the heart of the **ForceOne** compiler. In the process of type resolution, the algorithm inserts automatic widenings and narrowings at the appropriate places. Because overloading must be requested explicitly by the user, we expect there to be a large number of symbols that may be resolved directly from the symbol table. There are also a number of contexts in **ForceOne** that have a precisely defined type. The *test* in an `if` statement, for example, must be of type `bool`, and a *statement* delimited by ; must yield `void`. These strong contexts are used as anchors; the selection algorithm is invoked to determine the meaning of expressions between these anchors.

The selection algorithm operates on an expression-tree fragment rooted at an anchor whose leaves are either anchors or identifiers. If the sub-expression is correct, each symbol is resolved to a unique definition in the symbol table, and each edge in the tree may be labelled with the symbol table definition for an automatic conversion function. There are two ways in which the selection process may fail: the sub-expression may be inconsistent or the sub-expression may be ambiguous. In the case of inconsistency, we mark the set of two or more identifiers in the expression that are inconsistent. If the expression is ambiguous, we mark the identifiers that cannot be selected uniquely, and enumerate the possible definitions of each.

It is nearly impossible to avoid introducing ambiguity along with polymorphism and automatic conversions. We therefore define conversion costs and perform the selection so as to minimize this cost (the selection is considered ambiguous only if there are two alternatives that yield the minimum cost). The cost function is computed so as to implement the following rules.

1. If a polymorphic procedure and a type-specific procedure both match, the specific procedure is chosen.
2. A selection with no conversions is preferred to one that involves conversions.
3. Widenings are always preferable to narrowings: any number of widenings may be used to eliminate or reduce the number of narrowings.
4. Widenings are done as early as possible in the evaluation of an expression: it is always preferable to widen all the leaves of a sub-expression than to widen its result (this ensures that all intermediate results are calculated using the maximum precision in the expression).
5. Narrowings are done as late as possible in the evaluation of an expression (this ensures that intermediate results are calculated using the maximum precision in the expression).

The cost of a selection is a triple

```
<poly_cost, widen_cost, narrow_cost>   .
```

The cost of matching an actual parameter to a formal parameter with exactly the same type is `<0,0,0>`. The cost of matching an actual parameter to a formal parameter whose type contains a query parameter is `<1,0,0>`. The cost of widening an actual parameter to match a formal parameter is `<0,n,0>`, where `n` is the overall number of nodes in the expression tree representing the actual parameter. The cost of narrowing is `<0,0,m>` where `m` is the path length from the parameter to the root anchor. The operations on *cost* values are addition and comparison.

```
<a,b,c> + <d,e,f>   ≡   <a+d, b+e, c+f>
<a,b,c> < <d,e,f>   ≡   c < f or
                        c = f and b < e or
                        c = f and b = e and a < d
```

We may find the overall minimum cost using dynamic programming. The intermediate results are stored as attributes of nodes in the expression tree. For each subtree in the expression, we compute an attribute `typeset`, which is a set of pairs:

```
{ <type, cost> }   .
```

`type` represents a possible result type that the subexpression might yield; there is one pair in the set for each possible result type. `cost` is the minimal cost of achieving a result of type `type`.

There are three types of nodes in the tree: leaf anchor nodes, nodes representing overloaded identifiers, and interior nodes. Leaf nodes have a well defined type *t* and a cost of <0,0,0> to achieve *t* as a result type. Thus, all leaves are labelled with

typeset = { <*t* , <0,0,0>> }   .

Overloaded identifiers have many possible definitions; each definition d_i has a well defined type t_i. Thus we label such nodes with

typeset = { <t_i , <0,0,0>> }   .

An interior node always represents the application of a λ expression to an actual parameter list. The λ expression has the attribute λ.typeset. We evaluate the result types and costs of applying each element of λ.typeset to the actual parameter list.* The union of these result types and costs forms the attribute `typeset` for the node.

Once `typeset` has been computed for the root node, we can compare each `<type, cost>` against the type required by the anchor. If there is an exact match on `type`, the overall selection cost is `cost`. If a conversion is necessary, the overall cost is incremented by the cost of conversion. If no conversion is possible, or if `typeset` is empty, selection is impossible. As stated, the algorithm does not detect

* Normally, there is exactly one result type for a given λ type. However, a single λ type may yield different result types if its result type depends on a parameter. Furthermore, if there are any query parameters to the λ expression, we try binding each query parameter to each possible actual type.

ambiguity†.

The next step in the selection process is to mark the parse tree to indicate the appropriate definitions of overloaded symbols, and to insert necessary conversions. This labelling is accomplished by passing down the tree the attribute

```
actual_type = <type, cost>  .
```

When `actual_type` is passed down an edge of the tree, it is compared element by element with `typeset` of the lower node. It can easily be determined whether a conversion is required, and which element of `typeset` is the `actual_type` of the lower node. This information is marked in the tree. In order to propagate `actual_type` from an interior node to its children, we re-execute the matching algorithm, looking for the choice that yielded `actual_type`. This choice is marked; the selection becomes the value of `actual_type` that is passed to the λ expression, and the types of the formal parameters become the `actual_type` values that are passed to the actual parameters. When `actual_type` is passed down to a node representing an overloaded identifier, that reference is set to the definition whose type is `actual_type`.

3.5. The actual implementation

The **ForceOne** compiler is written in C and runs on the UNIX system. A **ForceOne** program is represented as a tree in the UNIX file system; VAX object code is generated and stored with the tree. In addition, the symbol tables and abstract syntax trees are stored in the tree. When a tree is recompiled, only those modules that have been modified are recompiled initially. Changes to declarations are detected by comparison with the old symbol table, and it is determined which, if any, modules are affected by the change in such a way as to require recompilation. Then these modules are recompiled.

The **ForceOne** compiler is approximately 12,000 lines of C code*: 3000 lines are devoted to lexical analysis, parsing, and building the abstract tree; 2200 lines are dedicated to the symbol table and overload algorithms; 2200 lines are for code generation; 1100 lines are devoted to debugging, error handling, and header files; the remaining 3500 lines implement the separate compilation facilities described above.

The front end of the compiler is complete; all the features described in this paper are handled. The code generator has been implemented only to the point necessary to run simple examples. Only a very small number of primitives are built-in: most functions result in calls to C library routines or must be implemented in **ForceOne** itself.

4. Comparison with related work

User-defined overloading is available in PL/I [PLI 76], Algol 68 [Wij 76], Ada [Ada 83], and C++ [Str 86]. In PL/I, a `generic` statement groups together a number of procedures under a common name. When the generic name is used, the appropriate definition is selected according to the types of the actual parameters. These types are not matched against the formal parameter types of the function definition, but are matched instead against abstract type specifications in the *generic* statement. After the selection is done, conversions are applied if necessary to match the formal and actual types. A disadvantage of the `generic` statement in PL/I is that all of the overloaded functions must be specified in one common place; there is no method to add another definition elsewhere. In Algol 68, operators may be defined by the user and may be overloaded. Selection is done by matching formal and actual types. Automatic conversions (which are normally available) are not done in matching operator parameters. Operators (unlike procedures) are not first-class in Algol 68; they may not be assigned or passed as parameters. In Ada, any operator or function may be overloaded by the user. In fact, there is no way of preventing functions that have the same name from overloading one another. The types of the parameters and the desired result type are used in selection. There are no automatic conversions in Ada. Neither functions nor operators are first-class; they may not be assigned or passed as parameters. The selection algorithm for **ForceOne** resembles those designed for Ada [cf. Aho 86].

† It is easily modified to detect ambiguity; a flag *ambiguous* is added to each *typeset* element. Whenever there are two equal-cost ways of yielding the same result, the flag is set.

* The source listing weighs 4.5 pounds, including the binder.

Overloaded operators in C++ are resolved bottom-up as in Algol 68, but user-defined conversions are allowed. At each step, a particular definition will be selected if no conversions are required to match its arguments. If no such definiton exists, there must be only one definition that can be matched by applying conversions to some of its arguments.* We know of no language that takes the type of the eventual result into account. In FORTRAN, for example, the assignment

```
X = 1 / 3
```

gives `X` (a `REAL` variable) the value 0.0. The equivalent expression in **ForceOne** would yield 0.333... (assuming the widening from `integer` to `real` were defined). While it has no user-defined conversions, PL/I has a wide variety of built-in automatic conversions: almost any type can be converted to any other type. The conversion rules in PL/I often yield unfortunate results (the integer 123 when converted to `CHAR(3)` yields `'   '`). We think these unexpected results may be attributed to the fact that many conversions will be applied transitively to match two types, and that the final result type is not taken into account in selecting the conversions.

Traditionally, polymorphism in compiled languages has been achieved either through loopholes in type checking, or by the use of macro processors. C [Ker 78] and PL/I are notable examples of languages that use both. Each mechanism has its own disadvantages, though both mechanisms involve leaving the domain of statically typed languages.

A more controlled approach is to have a type `union`. Functions that operate on `union` types are in a sense polymorphic, though they must enumerate all possible types on which they operate. Algol 68 provides an automatic `uniting` coercion that allows actual parameters to match the more general formal parameters. When the `union` type is used, a form of case statement called a `conformity clause` is used to determine the actual type of the parameter. Algol W [Sit 72] has unions only for *reference* types. EL1 [Weg 74] is a language that allows type manipulation and operations to be specified on objects whose type is known only at run time. However, these objects behave like unions: the set of possible types must be known in advance, and polymorphic operations are akin to *case* statements.

A large number of contemporary languages are based on the *class* concept of Simula 67 [Dah 70]. In Simula 67, a class describes a set of values and a set of operations. Each instance of the class has its own value, and its own set of operations. Simula 67 provides a polymorphic facility via class hierarchies. A general class may be defined that has a number of operations defined on it. These operations then automatically apply to any subclasses that are derived from this class. The superclass may also specify *virtual functions* that must be defined in the subclass. The Simula 67 class structure requires that all classes (and hence functions) be organized into a strict hierarchy, which severely restricts the nature of polymorphism that can be expressed in this way. Some languages (Taxis [Myl 80], for example) have "multiple inheritance" which allows a subclass to be derived from more than one superclass. Multiple inheritance ameliorates the restrictive nature of class hierarchies. Nevertheless, the philosophy of **ForceOne** is quite different; we do not require that all functions on an object be defined within the class of that object; we instead allow groupings of types and functions to be made at will. For example, in Simula 67, to write the polymorphic function `cube` (section 2.5), it would be necessary to create a class `cubable_objects` that contained the definition of `cube`. Then, to be able to apply `cube` to integers, we would have to modify the class `integer` to make it a subclass of `cubable_objects`.

Clu [Lis 77] reinforces the Simula 67 notion that a cluster (a.k.a. class, type) contains the definitions of all permissible functions. However, the notion of hierarchies is abandoned in Clu in favour of *generic* clusters. A generic cluster is a template that describes an infinite set of clusters. A generic cluster cannot be used directly; a specific member of the set must first be *instantiated* at compile time. Upon instantiation, constant parameter values are substituted and a new cluster (and a copy of each of its operations) is created. There is no way of "customizing" a cluster as in Simula 67. Clu supports overloading only in that each cluster may contain one definition of a particular function. When this function is applied the first parameter determines the cluster to which the function belongs. Ada provides polymorphism by special `generic` program units that resemble generic clusters.

* These rules could not handle the example of conversions between numeric types described in section 2.8.

Russell [Don 85] is a language that uses full run-time typing to achieve considerable expressive power: types are first-class values that are computed by procedures and may be stored in variables. A recent implementation of Russell [Boe 86] attempts to verify statically the type consistency of the program. The algorithm is complex and may fail (the problem is undecidable in general). With suitable restrictions* the algorithm is claimed to succeed in most cases. The notion that a type is the set of all permissible operations is essential to Russell – the run-time representation of a type is a list of procedures and a reference to a procedure is an index into this list.† The language Poly [Mat 85], based on Russell, has sacrificed some expressive power to simplify the language and its implementation; the essential philosophy is unchanged.

5. Final remarks

On the whole we are happy with **ForceOne**; it is remarkably adaptable to diverse applications and programming methodologies. Its use comes as a natural extension to programmers familiar with ordinary languages such as Pascal, C, or Ada. We believe that **ForceOne** has the power to express the essential elements of *object-oriented* programming, and of *functional* programming. We plan to examine real applications that are written using these methodologies to determine how easily they may be expressed in **ForceOne**. Adaptability is one payoff of the features of **ForceOne**; another is that it facilitates the creation of reusable libraries of very general purpose software.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada (grant A5485) and by **COGNOS** inc..

References

[Ada 83]
Reference Manual for the Programming Language Ada, U.S. Department of Defense, ANSI/MIL-STD-1815-A (1983)

[Aho 86]
Aho, A.V., Sethi, R. and Ullman J.D., *Compilers,* Addison Wesley (1986), 361-387.

[Boe 86]
Boehm, H. and Demers, A., *Implementing Russell* A.C.M. Sigplan Not. 21:7 (1986), 186-195.

[Cor 81]
Cormack, G.V., *Separate Compilation and New Language Features,* Ph.D. thesis, University of Manitoba (1981)

[Cor 83]
Cormack, G.V., *Extensions to Static Scoping,* A.C.M. Sigplan Not. 18:6 (1983), 187-191.

[Cor 85]
Cormack, G.V., *Zephyr,* unpublished (1985)

[Cor 86]
Cormack, G.V., Judd, M. and Wright, A.K., *Types are not Classes*, Univ. Waterloo CS-86-28 (1986)

[Dah 70]
Dahl, O.J., Myhrhaug, B. and Nygaard, K., *The Simula 67 Common Base Language, Norwegian Computing Centre S-22 (1970)*

[Don 85]
Donahue, J. and Demers, A., *Data Types are Values,* A.C.M. Trans. Prog. Lang. Syst. 7:3 (1985), 426-445.

* The rules are not easily stated; an understanding of the algorithm is necessary.
† In a sense, this model is a dual of that of EL1. In EL1, a procedure enumerates all types on which it might operate; in Russell a type is the enumeration of all procedures that might operate on it.

[Jud 85]
Judd, M., *A View of Types and Parameterization in Programming Languages,* M.Sc. thesis, McGill University (1985)

[Ker 78]
Kernighan, B.W. and Ritchie, D.M., *The C Programming Language,* Prentice-Hall (1978)

[Lec 84]
Leclerc, D., *Implementation Considerations for the Programming Language L,* M.Sc. thesis, McGill University (1984)

[Lis 77]
Liskov, B.H., Snyder, A., Atkinson, R. and Schaffert, C., *Abstraction Mechanisms in Clu,* Commun. A.C.M. 20:8 (1977), 564-576.

[Mat 85]
Matthews, D., *Poly and Standard ML,* A.C.M. Sigplan Not. 20:9 (1985), 52-76.

[Mil 78]
Milner, R., *A Theory of Type Polymorphism in Programming,* J. Computer Syst. Sci. 17 (1978), 348-375.

[Myl 80]
Mylopoulos, J., Bernstein P.A. and Wong, K.T., *A Language Facility for Designing Database-Intensive Applications,* A.C.M. Trans. Prog. Lang. Syst. 5:2 (1980), 185-207.

[Pas 80]
Specification for the Computer Programming Language Pascal, International Standards Organization, DP-7185 (1980)

[PLI 76]
O.S. PL/I Checkout and Optimizing Compilers: Language Reference Manual, IBM GC33-0009 (1976)

[Sit 72]
Sites, R.L., *Algol W Reference Manual,* Stanford University STAN-CS-71-230 (1972)

[Str 86]
Stroustrup, B., *The C++ Programming Language,* Addison Wesley (1986)

[Weg 74]
Wegbreit, B., *The Treatment of Data Types in EL1,* Commun. A.C.M. 17:5 (1974)

[Wij 76]
van Wijngaarden, A., et al., *Revised Report on the Algorithmic Language Algol 68,* Springer Verlag (1976)

[Wri 86]
Wright, A.K., *Reference manual for the language* **ForceOne**, unpublished (1986)

Appendix

Following is that part of the grammar for **ForceOne** relevant to declarations and polymorphism.

declaration	≡	*identifier* : *type* $\left[\ \texttt{==}\ \textit{expression}\ \right]$
type	≡	`ref` *type*
	≡	`routine` *type*
	≡	`[` *param_list* `]` *type*
	≡	*bound* `..` *bound*
	≡	`type`
	≡	`void`
	≡	$\left[\ ?\ \right]^{*}$ *identifier*
param_list	≡	*param* $\left[\ ,\ \textit{param}\ \right]^{*}$
param	≡	*identifier* : *type*
	≡	*type*
bound	≡	*integer*
	≡	$\left[\ ?\ \right]^{*}$ *identifier*