An Ad Hoc Approach to the Implementation of Polymorphism

R. MORRISON, A. DEARLE, R. C. H. CONNOR, and A. L. BROWN
University of St. Andrews

Polymorphic abstraction provides the ability to write programs that are independent of the form of the data over which they operate. There are a number of different categories of polymorphic expression—ad hoc and universal, which includes parametric and inclusion—all of which have many advantages in terms of code reuse and software economics. It has proved difficult to provide efficient implementations of polymorphism. Here, we address this problem and describe a new technique that can implement all forms of polymorphism, use a conventional machine architecture, and support nonuniform data representations. Furthermore, the method ensures that any extra cost of implementation applies to polymorphic forms only, and allows such polymorphic forms to persist over program invocations.

Categories and Subject Descriptors: D.1.0 [**Programming Techniques**]: General; D.3.3 [**Programming Languages**]: Language Constructs and Features; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs

General Terms: Design, Languages, Theory

1. INTRODUCTION

Polymorphism in a programming language is the ability to write programs that are independent of the form of the data objects that they manipulate. Thus, it provides an abstraction over the form of the data that is often categorized by type.

Polymorphism was first identified by Strachey [38] but did not receive serious investigation until the advent of languages such as ML [31], CLU [29], Russell [19], Poly [30], and some of the applicative languages such as Hope [8], Ponder [20], and Miranda [40]. More recently, the object-oriented methodology, which depends on polymorphic operations, and the generics of Ada [26] have raised the user community's awareness of the concept.

The advantages of polymorphic abstraction should be obvious in the context of software reuse [5, 35]. For example, a procedure to sort an array of

Authors' addresses: R Morrison, R. C. H. Connor, and A L. Brown, Department of Computational Science, University of St Andrews, North Haugh, St. Andrews, Fife, KY16 9SS, Scotland; A. Dearle, Department of Computer Science, University of Adelaide, Adelaide, South Australia 5000, Australia.

integers and another to sort an array of real numbers may be replaced by one polymorphic procedure that will work for all relevant types. In a large system, this can greatly reduce the amount of code that has to be written and maintained.

Cardelli and Wegner [12] give a classification that divides polymorphism into two categories called *ad hoc* and *universal*. According to Strachey, in ad hoc polymorphism there is no single systematic way in which to determine the type of the result of a function from the type of its arguments. Burstall and Lampson [7] give an operational definition, stating that in ad hoc polymorphism the code executed depends on the type of the arguments. As an example, the **print** function executes differently when evaluating

print 42

than when evaluating

print "42"

The **print** function is polymorphic since it is defined over more than one type, but the code executed may be different in each case. Most ad hoc polymorphic functions are "built in," that is, supplied by the system, but Keas [28] and Wadler and Blott [42] have shown how user-defined ad hoc polymorphism also may be utilized.

Universal polymorphism has two forms: *parametric* polymorphism and *inclusion* polymorphism. In parametric polymorphism, a polymorphic function has an explicit or implicit type parameter that determines the type of the argument for each application of that function. In inclusion polymorphism, an object may be viewed as belonging to many different types (classes) that need not be disjoint. Cardelli and Wegner point out that the two forms of polymorphism are not disjoint but are sufficiently different to deserve different names [12].

Parametric polymorphism describes the polymorphism found in ML and its derivatives, whereas inclusion polymorphism is the style of polymorphism found in object-oriented languages such as Simula67 [4]. An interesting hybrid may be found in the database programming language Galileo [1], which is a derivative of ML but utilizes inclusion polymorphism to implement part of the Semantic Data Model [23]. Cardelli [11] has shown separately how the parametric and inclusion forms of polymorphism may be integrated, as bounded quantification, to yield forms of abstraction not available to either one.

Some languages allow both parametric polymorphism and ad hoc polymorphism to coexist. For example, in Napier88 [34], parametric polymorphic functions may be defined, but others such as equality, which execute different code for different types, are also supported. In Ada, the instantiation of generics allows for both ad hoc and parametric polymorphism. Finally, Hudak et al. [25] allow both ad hoc and parametric polymorphism to coexist while retaining the power of type inference.

Much of the published work on polymorphism concentrates on the design of polymorphic forms of expression and their utility to a particular application

area [12, 22, 33]. Other published work concentrates on the algorithms required to write polymorphic type checkers [10, 13, 17, 21, 32]. In particular, much work has been done on type inferencing, that is, the ability to infer the type of polymorphic expressions from a static scan of their usage. Such an ability is sometimes seen as the most important aspect of polymorphism, as it frees the programmer from the necessity of writing down complicated type expressions to augment the executable code. A good example is given in [37] where the type of natural join is inferred. Whether or not the type is inferred, there remains the problem of generating code from the abstract description of the program within the compiler.

In this paper we concentrate on machine architectures and code generation techniques required to support polymorphism. We do this by reviewing some known techniques and then by suggesting a new method that has been used to implement the language Napier88. This utilizes the block retention architecture of the Persistent Abstract Machine [6, 16] and allows polymorphic expressions to manipulate values of nonuniform sizes. Other features of the method are that only polymorphic expressions, and not others, pay the extra cost of genericity and that the technique is suitable for implementation on traditional architectures. Finally, the system also supports the persistence of polymorphic values.

2. IMPLEMENTATIONS OF POLYMORPHISM

In polymorphic expressions, a computation is described independently of the type of the data being manipulated. This polymorphic description can be at several levels of abstraction: the computation can be expressed at the level of source code, machine code, μ-code, or any combination of these. Therefore, the meaning of polymorphism in any system is interpreted with respect to a level of abstraction. For this reason, we introduce some definitions to clarify our text.

There are three extremities in implementation techniques for polymorphism that depend on the level of abstraction at which the polymorphism is expressed. For any polymorphism to exist, the source code of a program must exhibit expressions that are independent of the type of the data. If this is the limit of the polymorphic expression, then we use the term *textual polymorphism* to categorize it. For example, in Ada generics, only the source code displays polymorphism since different machine code may be generated for each different instantiation type. Similarly, languages that exhibit ad hoc polymorphism may execute the same code for different types at the source level, but may in fact execute different code at the machine level.

A second perhaps more pure form of polymorphism is found in systems where both the source code and machine code for polymorphic expressions are independent of the data types manipulated. This we term *uniform polymorphism*. In ML, which uses uniform polymorphism, all data have a uniform representation, and the same code is executed at both the source and machine level for polymorphic expressions regardless of the type of the data

```
let id = proc [t] (x : t → t) ; x

let forty_two_int = id [int] (42)

let forty_two_real = id [real] (42.0)

type car is structure (doors : int ; paint : string)

let this_car = car (4, "blue")

let same_car = id [car] (this_car)
```

Fig. 1. The identity procedure and calls.

being manipulated. As we will see, this has implications for the efficient representation of data, including nonpolymorphic objects.

Finally, other systems have source and machine code uniformity but nonuniform store representations. This means that machine instructions must take account of the differing store formats. Thus, the machine code for a polymorphic expression takes account of type information when it is executed. Many of the object-oriented languages are implemented in this manner, and all systems that depend on a tagged architecture also fall into this last category, which we term *tagged polymorphism*.

We explore each of the above implementation categories in turn since each yields different trade-offs for the efficient implementation of polymorphism. It should be remembered that textual, uniform, and tagged polymorphism represent extremities in the range of implementation techniques. Most languages will be implemented by a mixture of techniques within this range. The reason for exploring the extremities is to expose the limits of each technique and, therefore, the trade-offs available to the implementor.

First, we introduce the syntax of Napier88, in which the examples are written.

2.1 Nomenclature

As an introductory example, we present the polymorphic identity procedure written in Napier88, which is given in Figure 1. We will explain this example in detail for those unfamiliar with the syntax of Napier88 and for the benefit of later examples.

The polymorphic identity procedure is written as

```
let id = proc [t] (x : t → t); x
```

That is, the identifier *id* is declared, by the reserved word **let**, to be a procedure that is quantified by t, written [t], that takes a parameter x of type t and yields a value of type t. The body of the procedure is the expression x, which when evaluated yields the result.

This procedure may be called by

```
id [int] (42)
```

which specializes the quantifier *t* in procedure *id* to be integer, written **int**, and applies the procedure to the value 42. The result is, of course, the integer 42. Notice that, for the moment at least, there is no type inferencing [32] on the calling of procedures and that the specialization type is written explicitly.

The procedure may also be specialized for real values. For example,

```
id [real] (42.0)
```

yields, in the same manner, the real number 42.0.

Finally, more complicated objects may also be used. For example,

```
type car is structure (doors : int ; paint : string)
```

defines a type *car* to be the labeled Cartesian cross product, called a structure, containing a label *doors* of type integer and another *paint* of type string. Such a value may be created and bound to an identifier by the declaration

```
let this_car = car (4, "blue")
```

Notice that the type identifier *car* also serves as the constructor function identifier. The identity procedure may be used to operate on *this_car* as follows:

```
id [car] (this_car)
```

which will return the value *this_car*.

For now we will use procedures as our form of polymorphic expression and show later how other forms such as abstract data types (objects) may be implemented.

2.2 Textual Polymorphism

In textual polymorphism, the polymorphic uniformity applies to the source code only. Thus, the compiler may generate code specific to the specialization of the polymorphic procedure. For example, the calls of the identity procedure *id*

```
id [int] (42),
id [real] (42.0),
```

and

```
id [car] (car (4, "blue"))
```

may all execute different versions of the code for *id*. This would be the case if 42, 42.0, and *car* (*4*, "*blue*") were all represented by different store formats in the machine. Indeed, the early examples in this section are deliberately kept simple for clarity and therefore look like prime candidates for implementation by uniform polymorphism. To overcome this, for the moment, the reader should imagine that each data type has a different store representation necessitating different code to be executed for each call that is specialized to a different type.

One method of implementing textual polymorphism is for the compiler to translate the polymorphic procedures into an intermediate form. When a

```
let triples = proc [d, e, f] (D : d ; E : e ; F : f )
```

Fig. 2. A polymorphic procedure with multiple quantifiers.

specialization of the polymorphic procedure is compiled, the intermediate form is used to generate code specific to the specialization type. This generated code may be in the same language, in which case the compiler will then compile the generated code [37]. Alternatively, the intermediate form can be used to generate specific machine code directly.

The major advantage of the textual polymorphism technique is that the compiler can produce optimum code using optimum representations of data. That is, the code can be tailored to the type of the specialization, producing the required run-time speed and space overhead. A second advantage is that the technique can be used to implement ad hoc polymorphism since it utilizes an ad hoc implementation mechanism that can generate code specific to the specialization.

Within a compilation unit, a bound can be placed on the number of specialized forms of the polymorphic procedure that may be generated. In the upper limit, a separate form for each specialization type may be required. In the lower limit, there is only one form, and the method reverts to the uniform polymorphism of the next section. In practice, in any compilation unit, there is a fixed number of data types, some of which may share representations. This leads to a fixed upper limit on the number of possible specialized forms of any particular polymorphic procedure, which is the number of representations of the quantifier types raised to the power of the number of type quantifiers. This is highlighted in Figure 2.

In Figure 2 the procedure *triples* is quantified by three types d, e, and f, and takes three parameters D, E, and F of those types, respectively. The procedure is of type void and, therefore, does not require specification of its result type. The procedure body is left unspecified since it is not of interest to our argument.

The compiler may have to generate every possible specialized form for all three quantifier types. If there are n possible forms for the quantifiers and m quantifiers, then there are n to the power m possible forms for the polymorphic procedure. In this case it is n^3, where n is the number of possible representations of the quantifier types.

Of course, the above gives a bound for the number of possible specialized forms of a polymorphic procedure, but it should be noted that for any compilation unit the compiler only has to generate one specialized form of the procedure for each static call. Also, many of the calls will have the same specialization type representation, allowing them to share the implementation. Thus, the number of specialized forms of a polymorphic procedure in any compilation unit is determined by the variety of call specializations. This is, of course, not the case with separate compilation where the number of specializations is not known statically. Indeed, it should be obvious that with textual polymorphism a polymorphic procedure may not be applied outside the context of its compilation unit. However, Figure 3 demonstrates how even the above simple technique may become complicated.

```
let first = proc [a, b] (A : a ; B : b)
                ...
let second = proc [s, t] (S : s ; T : t)
                if <condition> then first [s, t] (S, T) else first [t, s] (T, S)

second [int, bool] (43, true)
second [real, int] (42.999, 43)
```

Fig 3. Multiplicative expansion in code.

```
let third = proc [s] (S   s ; A : proc [t] (t → t) → s)
                A [s] (S)

let id = proc [u] (x : u → u) ; x

let forty_two = third [int] (42, id)
```

Fig. 4. Passing polymorphic procedures as parameters.

Both of the procedures, *first* and *second*, have two type quantifiers. For each call of *second*, there are two possible calls of *first*, and therefore, two specializations may be required. They are not required in the trivial case where the types s and t are the same. This second-order effect is multiplicative and may escalate to a large number of specializations, all of which are necessary. The total number of specialized forms of a polymorphic procedure for any one call may be found by multiplying the number of different specializations of each possible procedure in the call chain. This may become very large and involves calculating the transitive closure of calls in a program. For example, if procedure *first* had three different specializations of another polymorphic procedure then six versions of that procedure are required for each call of *second*.

This problem is also present when polymorphic procedures are allowed to be passed as parameters. Figure 4 gives such an example.

In Figure 4 procedure *third* is quantified by s and takes a parameter S of the quantifier type s and another A, which is a polymorphic procedure of type

proc [t] (t → t)

The body of the procedure *third* calls the polymorphic procedure A with the specialization type s and the value S. The call of *third* specializes the quantifier to **int** and passes the polymorphic procedure *id* and the integer value 42. For each specialization of *third*, there is also a specialization of A

```
let either = if <condition> then proc [t] (a, b : t → t) ; a
                            else proc [t] (a, b : t → t) ; b
...

let two = either [int] (23, 12)

let three = either [real] (-0.2, -12.123)
```

Fig. 5. First-class polymorphic procedures.

that has to be found from the call of *third*. This expansion in code is again multiplicative.

The translation method, for textual polymorphism described above, can be used for the above examples but does not work where the polymorphic procedures are first-class data objects or may be stored in a persistent store.

Languages such as Napier88 allow first-class polymorphic procedures [2]. These procedures may be stored in data structures, assigned and substituted for one another if they have the same type. Figure 5 gives such an example.

In Figure 5 the procedure *either* is initialized to one of two polymorphic procedures, depending on the value of ⟨condition⟩ when it is executed. When compiling the expression *either* [**int**] (23, 12), the compiler cannot determine which of the two procedures is being called, since it depends on the dynamic flow of the program. The same is true for the call *either* [**real**] (-0.2, -12.123). The determination of which of the two procedures to call can only be performed at run time.

To implement the above, by textual polymorphism, the specialization of the procedure should be performed at run time during the call. One possible implementation technique is for the compiler to generate every possible form of the procedure statically and for the run-time system to pass around pointers to this code. On specialization, the type may be used to select the particular version to call. However, we have already demonstrated that the number of possible forms may become very large. In long-lived systems, the storage requirement may become great.

An alternative implementation technique is to pass around a pointer to the source code and for the compiler to be called dynamically on specialization [18]. However, this will make the call of the polymorphic procedure slow. An optimization is to specialize the procedure on the first call for every type and to retain the specialized code for latter applications. This is equivalent to memoizing the specialization.

Neither of the above two implementation techniques is practical for reasons of space and time overhead, respectively.

Since this method is unable to implement first-class polymorphic procedures efficiently, it is also unsuitable for storing polymorphic procedures in a persistent store since they may also be specialized dynamically.

To summarize, the advantages of this technique are that

—it can produce optimum code for each application of the polymorphic procedure,

—it can implement nonuniform representations of data, and
—it can implement ad hoc as well as universal polymorphism.

The disadvantages are that

—the amount of machine code generated for each polymorphic form may be large,
—it is unsuitable for the implementation of first-class polymorphic procedures, and
—it is unsuitable for supporting the persistence of polymorphic procedures.

The generics of Ada are a special case of this implementation technique, and the above perhaps explains why generics are not first-class data objects in that language.

2.3 Uniform Polymorphism

In the uniform polymorphism implementation technique, the uniformity is present at both the source code level and machine code level. To implement polymorphic procedures, such as *id* above, the compiler must generate code that will execute in exactly the same manner for all calls of the procedure. A consequence of this uniformity is that all data types must have the same store representation for this uniform code to execute correctly. Thus, in Figure 1 objects of type integer, real, and *car* must have the same store representation. This means that, for all data types, there is a fixed stack element size and the same fixed size for components of heap objects. This second requirement is necessary since the addresses of components of heap objects must be the same irrespective of the concrete types of the fields. That is, a particular field must always be at the same address.

Notice, however, that this fixed store representation may not be the optimum nor even the desired representation of some types. For example, for the calls

```
id [int] (42)
```

and

```
id [real] (42.0)
```

above, if a uniform size of one word for integers is used, it becomes difficult to implement double-length reals efficiently. On the other hand, if double-length reals are implemented efficiently then the implementation of integers may be nonoptimum.

The second disadvantage of this technique is also a consequence of the uniformity of data. In order to implement multiunit data such as in the call

```
id [car] (car (4, "blue"))
```

above, the compiler must group the elements together and refer to them by a pointer. This is because a fixed element size must be used and all different sized structures must be uniformly represented. By making the pointer the uniform size, all pointers may now be manipulated in the same manner by

polymorphic procedures. Of course, the maximum size of all objects could always be used as the fixed element size, but this would be very wasteful of storage. It may also be unknown for separately prepared program and data [3].

A consequence of the fact that pointers must be found for garbage collection is that all data objects must now be represented by pointers. Thus, for scalar items such as integers and reals, there is an extra level of indirection in their implementation. A clever optimization of this appears in the Functional Abstract Machine [9], which is used to support ML (D. McQueen, private communication, Second International Workshop on Database Programming Languages, Salishan Lodge, Oreg., 1989) and Galileo. A one-bit tag is used in every data item to distinguish pointers from nonpointers. Since the languages are statically typed, the compiler always knows whether a pointer or a nonpointer is being manipulated, except in a polymorphic procedure where both are manipulated in the same manner anyway. Thus, only the garbage collector needs to use the tag bit. The minor disadvantage of this scheme is that one bit is used for the tag, thereby complicating arithmetic and reducing the address space.

To summarize, there are two major advantages of this technique. They are that

—it is relatively easy to implement since the compiler generates uniform code for each form; and

—there is only one copy of the machine code for polymorphic forms and it is, therefore, efficient in code space.

There are, however, a number of disadvantages to the technique. They are that

—uniformity is imposed on all data types without regard to optimal representation, and

—all objects pay the price of the system being polymorphic.

This form of polymorphism is the kind found in ML, although it may not always be implemented in this manner. Wadler and Blott [42] have shown how to extend this technique, at the language level, to allow it to be used in the implementation of ad hoc polymorphism. They give examples of some arithmetic operators and equality. However, this method may also be used to support assignment when regarded as a polymorphic operator. This would allow their technique to be used for nonapplicative languages. However, their implementation effectively uses class tags and is more appropriately classified in the next section.

2.4 Tagged Polymorphism

Tagged polymorphism displays uniformity at the source code level and at the machine code level. However, the data objects and the manner in which the generated code instructions operate over them may be nonuniform. Effectively, every data item is tagged with its type, and the generated code uses this tag to determine how to execute the type-dependent instructions.

The best-known implementation of this technique is the use of a tagged architecture such as the Burrough's B6500 [24]. In that, there are some polymorphic operations such as plus, minus, times, equal, and not equal that operate according to the data tag. For example, both integers and reals have a times operation defined over them. On execution, the machine instruction inspects the tag and performs either integer or real multiplication. Indeed, the real numbers themselves may be single- or double-precision, constituting a further variation.

Some object-oriented languages use a similar technique to implement multiple inheritance (inclusion polymorphism). In this, each object contains an address map for its methods. The name of a method being selected is used as a key to the address map. The static machine code for searching an address map may be the same for all methods. However, the address maps for different objects will be different, and thus, searching for the same method in different objects will cause different dynamic sequences of code to be executed. The address map is effectively a tag.

Tagged polymorphism is essentially a technique for implementing "built-in" ad hoc polymorphic operations. The machine instructions that look at the tag constitute these polymorphic operations. However, if the data types of a particular language can be mapped onto the tags, then the mechanism can be used to implement a mixture of parametric and ad hoc polymorphism. Indeed, it can even be used to implement more dynamic forms of polymorphism like that found in languages such as SASL [39] and Smalltalk [22].

There is always difficulty in mapping an infinite type system onto a finite tag representation. In most systems, however, many different types will share representations, and an encoding of the representation can be used as a tag, thereby alleviating the difficulty.

The advantages of the technique are that

—compact code is generated for polymorphic forms;
—it can implement ad hoc, inclusion, and parametric polymorphism; and
—it can operate on nonuniform data.

The major disadvantages of this technique are that

—the polymorphic operations are built in,
—the tagging must be efficient, and
—all data types must be tagged all of the time.

The technique depends on efficient tagging. However, hardware tagging is not generally available, and software tagging is too expensive except where the instruction sequence to be executed is large compared to the tag interrogation sequence, as in object-oriented languages. van Vliet and Gladney [41] performed an evaluation of tagging by writing two sample applications in both Smalltalk and PL/1. They found that most of the dynamic lookup of methods in Smalltalk could be performed statically in PL/1. The remaining ones could be implemented using tags. By mixing tags with nontagged items in the same language, the tags could be used to produce flexibility at the

required point. Since only some items required tagging, the implementation became very efficient.

3. AN AD HOC APPROACH

In this section we describe the method used to implement polymorphic procedures in Napier88. Special difficulties arise because of the availability of first-class polymorphic procedures, specialization without call, and the persistence of polymorphic forms. Specialization without call occurs where a polymorphic procedure is specialized but not called. An example will be given later. The technique is, however, general and may be used to implement other polymorphic languages. When used in combination with other implementation techniques, an ad hoc implementation strategy can be evolved.

Ideally, we would like one implementation technique that supports all forms of polymorphism. However, given that all the implementation methods described in Section 2 have major shortcomings, it is unlikely that this goal can be achieved. When judging the implementation strategies, we use the following criteria:

—It should implement all forms of polymorphism,
—it should be implementable on conventional architectures,
—it should support nonuniform data representations efficiently,
—it should support first-class polymorphic procedures,
—it should support the persistence of polymorphic forms,
—only polymorphic procedures should pay any extra implementation costs, and
—the polymorphic code should be optimally compact and fast.

We begin by describing the machine architecture for implementing polymorphism in Napier88. We then describe how this may be used with other techniques.

3.1 The Napier88 Approach

Napier88 utilizes a variant of tagged polymorphism as its implementation technique. However, since the language supports first-class procedures and specialization without call in a persistent environment, the tagging is performed by using procedure closures to capture type information.

Only polymorphic procedures need to be uniform in the manner in which they operate. Even then, it is only the polymorphic expressions within them that require uniformity. Outside the polymorphic procedures, this uniformity may be neither efficient nor desirable. It is, therefore, possible to implement a system whereby any data object of a quantifier type is coerced to a uniform polymorphic form on entering a polymorphic procedure and coerced back when leaving. Within the procedure, the object of the quantifier type is also manipulated in its uniform representation. It should be noted that it is only necessary to convert the parameters and results of the particular quantified

```
let first = proc [t] (a, b : t → t) ; a

let either = if <condition> then first [int] else proc (a, b . int → int) ; b

let two = either (23, 12)
```

Fig. 6 First-class procedures and specialization without call.

Fig. 7. A concrete actual parameter passed to a quantified formal parameter

```
let id = proc [t] (x : t → t) ; x

let two = id [int] (2)
```

type, and not others. This basic idea is similar to the one discovered and reported in [41].

3.1.1 *Points of Conversion.* The combination of first-class procedures and specialization without call in a programming language ensures that the compiler cannot determine statically whether a procedure being called is polymorphic or not. Figure 6 illustrates this.

In Figure 6 the procedure *either* is of type **proc (int → int)**. It is initialized to the value of procedure *first* specialized to integer or to the procedure defined after the **else**. Notice that the procedure *first* is specialized but not called in the **if** clause.

The call *either* (23, 12) could be to the specialized polymorphic procedure *first* or to the nonpolymorphic procedure. In general, the compiler cannot determine statically whether or not a polymorphic procedure is being called. Thus, conversion code to and from polymorphic representations cannot, in general, be performed before the call. It must, therefore, be the responsibility of the polymorphic procedure to convert any parameters of quantifier types inside the procedure and reconvert them before returning.

When passing parameters to a polymorphic procedure, four different cases of parameter passing are of interest. The formal and actual parameters may have concrete or quantifier types and representations. The cases are

—a concrete actual parameter is passed to a concrete formal parameter,
—a concrete actual parameter is passed to a quantified formal parameter,
—a quantified actual parameter is passed to a concrete formal parameter, and
—a quantified actual parameter is passed to a quantified formal parameter.

The first case is given for completeness, but since there is no polymorphism involved, the compiler can generate nonpolymorphic code. Figure 7 gives an example of the second case, where a concrete actual parameter is passed to a quantified formal parameter.

For each formal parameter of quantifier type, the corresponding value for each actual parameter must be converted inside the polymorphic procedure for use and the result, if of a quantifier type, reconverted on exit. Inside the procedure they can be manipulated in their polymorphic form.

```
let int_id = proc (x : int → int) ; x
let p = proc [t] (x : t ; y : proc (t → t) → t) ; y (x)
let three = p [int] (3, int_id)
```

Fig. 8. A quantified actual parameter passed to a concrete formal parameter.

Fig. 9. Passing a quantified actual parameter to a quantified formal parameter.

```
let q = proc [t] (x : t → t) ; id [t] (x)
```

```
type thing [t] is structure (First : t)
let make_thing = proc [a] (A : a → thing [a]) ; thing [a] (A)
let first_thing = make_thing [int] (42)
```

Fig. 10. Quantified values in data structures.

Since the polymorphic procedure has uniform code for all types, calling the procedure with an integer or a *car* only requires conversion code to depend on the original type. All other code is uniform.

Figure 8 shows the third case, where a quantified actual parameter is passed to a concrete formal parameter. The procedure p is quantified by t and takes a parameter x of that type and another y, which is a procedure from $t \rightarrow t$. It returns a value of type t that is calculated by applying y to x. On the call of y, the value 3, which has a quantifier representation inside p, must be reconverted for passing to y, which is the nonpolymorphic procedure *int_id*. On return, the result is converted to quantifier form.

Figure 9 shows the fourth case, where a quantified actual parameter is passed to a quantified formal parameter. The polymorphic procedure *id* is called with the value x. Unfortunately, *id* expects to convert concrete values on entry, and so x must be made concrete for the call and converted back to quantifier form on the return.

There is one final problem that must be enumerated before an implementation can be discussed. It is illustrated in Figure 10.

In Figure 10 type *thing* is parameterized by t. It is a structure with a field *First* of type t. Procedure *make_thing* creates a *thing* parameterized by its quantifier type a and returns it as the result. Outside the polymorphic procedure, *first_thing* is expected to be a structure with an integer of value 42 in it. Thus, when the structure is created it should be constructed in this form. Furthermore, it is possible for such objects to have different types simultaneously when the structure is passed into a polymorphic procedure. Within a polymorphic procedure, the field will have its quantifier type, but outside it will have its concrete type. The field address calculation must work in both cases but may be performed by different mechanisms.

In general, placing quantifier values into and getting them out of data structures are special cases of procedure return and parameter passing. The same rules apply. That is, only concrete representations may be used in data structures, and when they are selected from or assigned to in a polymorphic procedure, a conversion takes place.

```
let random = begin
                let seed := 2111

                proc ( → int)
                begin
                    seed := (519 * seed) div 8192
                    seed
                end
            end
```

Fig 11. Block retention.

To summarize, because of first-class procedures and specialization without call, the conversion of values of quantifier types must be performed within the polymorphic procedures. This is true for parameters and result values. We have enumerated the possible cases for parameter passing and have shown how the conversion is also necessary for data structures that may be accessed by the polymorphic procedure.

3.2 The Napier88 Block Retention Architecture

The implementation of polymorphism in Napier88 utilizes the block retention nature of the language. The block retention architecture is necessary to support higher order procedures [27], as in Figure 11.

In languages without block retention, the space for values that are declared within the block may be reclaimed on exit from the block. However, in the above, the block contains the declaration of a variable *seed* and has a value that is a procedure. The space for the block that contains *seed* cannot be reclaimed on exit since *seed* is in the closure of the procedure now called *random*. The block must be retained for *random* to operate correctly.

To demonstrate the Napier88 implementation of polymorphism, we illustrate the implementation for nonuniform sized values, for example, by representing integers by one word and reals by two words.

Every polymorphic procedure in Napier88 is compiled into another in which the type parameter is represented by an integer in an outer level procedure. This is illustrated by the following example,

```
let id = proc [t] (x : t → t) ; x
```

which is compiled into

```
let id = proc (tTag : int → proc (α → α))
  proc (x : α → α) ; x
```

The outer procedure forms an envelope containing the integer tag around the procedure with the executable code. That is, *id* is now the envelope procedure that takes as a parameter an integer *tTag*, which is an encoding of the quantifier's specialization type. *tTag* varies for each call. The result of

calling *id* is a procedure that takes as a parameter x a value of some type and returns it as the result. Thus, *id* is called twice, once for specialization and then with the specialized value. For the moment, the type α is not important. It is, however, at any specialization, the concrete type of the quantified type.

The tag determines the representation of the type of the parameter at run time. Thus, the call

let int_id = id [**int**]

will call the envelope procedure of type

proc (tag : **int** → **proc** ($\alpha \rightarrow \alpha$))

with the integer tag for type integer. Thus, *int_id* is now the procedure

proc (x : $\alpha \rightarrow \alpha$) ; x

with the type tag encapsulated in its closure. Notice that this procedure must execute in the same manner for all types. It does so by executing the same code, since there is only one copy of the code, but using the tag to discriminate types where the instruction depends on type. This particular procedure must convert its parameter to quantifier form depending on the type's tag and reconvert the result before returning the value. Thus, the subsequent call

int_id (32)

will cause the procedure to convert 32 to the uniform quantifier form on entry, according to its type representation tag, and reconvert on exit.

The call

id [**int**] (32)

is directly compiled into two calls, one to the envelope procedure with the type tag, followed by one to the result of this call with the integer value 32.

On entry to the procedure, the instruction *convertToPoly* is executed using the representation tag in the closure. The *convertToPoly* instruction converts the concrete representation of the data into a polymorphic one. The tag determines the concrete form, and the polymorphic one is uniform. On exit from the procedure, the *convertFromPoly* instruction is executed, again using the given tag to convert from the polymorphic form to the concrete one. These two built-in instructions operate differently for different tags.

The call

id [**real**] (42.0)

will execute the same code as *int_id* but with a different tag in its closure to determine the conversion to and from the polymorphic form. Finally,

id [car] (car (4, "blue"))

will execute the same code again but with yet another tag. Thus, there is one version of the code that executes differently for every tag.

It should be emphasized again that there is only one version of the code for the procedure

proc (x : $\alpha \rightarrow \alpha$) ; x

but that it may be bound to many envelopes of type

proc (tag : **int** → **proc** ($\alpha \rightarrow \alpha$))

The efficiency of this technique depends on the number of polymorphic operations and the efficiency of the tagging. Unlike the tagged machine architecture solution, only quantifier values are tagged. Also, only polymorphic procedures require tagged code. Thus, the tagging overhead is small for polymorphic code and nonexistent for nonpolymorphic code.

In Napier88 the type system describes an infinite number of data types. However, as pointed out earlier, the number of data type representations in most languages is finite since the compiler uses a finite number of store representations. In Napier88 it has proved possible to map this infinite type system into seven distinct machine tags. The technique does not, however, depend on this but depends only on that the type system can be represented by tags. The polymorphic operations *convertToPoly* and *convertFromPoly* should be regarded as ad hoc polymorphic procedures called with a type tag themselves. They are, however, built into the system and, therefore, ground the recursion on the use of tags.

3.3 Implementation of Data Structures

As described earlier, data structures that are manipulated by polymorphic procedures and contain fields of a quantifier type yield examples where the extent of the quantifier value may escape the scope of the procedure. The problem arises whenever the extent of an object containing values of quantifier types and manipulated by the polymorphic procedure is different from the scope of the procedure.

The solution proposed is that all data structures contain only representations of fields of nonpolymorphic type. Thus, where the fields are manipulated by polymorphic procedures, they must be converted for use within the procedure and reconverted when placed in the data structure. This is exactly the same rule as for parameters and returned values.

The given solution is necessary since the data structure may be accessed from different points in the program and, therefore, must have one fixed representation.

There are two cases where a polymorphic procedure may manipulate a value of a quantifier type that is part of a data structure: (1) when the data structure is passed as a parameter and (2) when the data structure is created within the procedure and returned as its value. Figure 12 illustrates the first case.

In Figure 12 the procedure *findSize* is quantified by types *s* and *t*. It takes, as a parameter, a structure with fields *age* and *size* with types *s* and *t*, respectively, and returns a value of type *t*, which is the *size* field of the structure.

```
let findSize = proc [s, t] (A : structure (age : s ; size : t) → t) ; A (size)
let intThree = findSize [real, int] (struct (age = 42.0, size = 3))
let stringThree = findSize [int, string] (struct (age = 42, size = "Three"))
```

Fig. 12. Passing structures with quantified fields to polymorphic procedures.

```
let findSize = proc (sTag, tTag, ageOffset, sizeOffset : int
                → proc (A : structure (age : α ; size : β) → β))
                    proc (A : structure (age : α ; size : β) → β)
                        A (sizeOffset)
```

Fig. 13. Compilation of Figure 12.

The two calls of the procedure specialize it to different types and pass structures of different types. Thus, if the types have different store representations, it is impossible to generate the address of the fields within the data structure, in the polymorphic procedure, statically. They may be different for every call.

The solution is an extension of the method already described. Although the compiler cannot calculate the field offset addresses within the procedure, it can do so at the point of the call. Thus, the call can pass the field offset addresses into the procedure, which can then use them to index the structure. The procedure that holds the quantifier type tags can also hold the field offset addresses in its closure. The *findSize* procedure above can now be compiled as in Figure 13.

In Figure 13 the *findSize* procedure is compiled into one that takes the two specialization tags as parameters along with the field offset addresses for the *age* and *size* fields. When specialized, the procedure returns another procedure that takes a structure and returns the *size* field value. This second procedure uses the value *sizeOffset* within its closure to index into the particular structure for the particular *size* field.

Each specialization of the types yields different tags and offsets. The call of the procedure must provide the field offset address values. If we assume that integers occupy one field, strings two, and reals three and that the structures fields are organized in their declared order, the two calls will now be compiled as

```
let intThree = findSize (realTag, intTag, 0, 3) (struct (age = 42.0, size = 3))
let stringThree = findSize (intTag, stringTag, 0, 1) (struct (age = 42, size =
  "Three"))
```

There is only one polymorphic form for each data structure and different offset addresses for each procedure call. The compiler knows either the offset address of the field or the address of the offset address of the field and can

```
let mkPair = proc [s, t] (first : s ; second : t → structure (fst : s ; snd : t))
                struct (fst = first, snd = second))
```

Fig. 14. A polymorphic procedure that creates a data structure with quantified fields.

generate the correct code according to circumstances, which include passing the data structure on to a further polymorphic procedure.

A second addressing problem arises where the data structure is created within the polymorphic procedure. Figure 14 gives such an example, where the procedure *mkPair* is quantified by the types *s* and *t* and takes parameters *first* and *second* of these types, respectively. The procedure creates a structure and returns it as the result. This structure must be of the same form as if it were created outside the procedure using the specialization types.

The addressing problem occurs since neither the overall size of the structure nor the field offset addresses are known at compile time. They depend on the representation sizes of the specialization types.

Again it is possible to modify the Napier88 technique to handle this case. Either the call can generate the size and field offset addresses in the manner described above, or code can be generated in the envelope procedure to calculate this information on a call and leave it in the closure of the polymorphic form as local declarations rather than the parameters. This second method is used since some of the structures used inside polymorphic procedures will be totally encapsulated by them and the size information may not be available at the time of the call.

Thus, *mkPair* would compile as shown in Figure 15.

If we assume again that integers occupy one field, strings two, and reals three and that structures are organized in their declared order, then the missing procedure *typeSize*, which is built in, may be encoded as follows:

```
let typeSize = proc (tag : int → int)
                 case tag of
                   intTag    : 1
                   stringTag : 2
                   realTag   : 3
                 default     : 0
```

Thus, if the call

```
mkPair [real, string] (42.0, "ronald")
```

was made it would cause *realTag* and *stringTag* to be substituted for *sTag* and *tTag*, respectively, and would cause the internal values *sndOffset* and *structSize* to be 3 and 5, respectively. Thus, when the resultant procedure is executed, 42.0 will be placed in the structure at address 0 and "ronald" at address 3. The overall size of the structure is 5.

The important point about this discussion is not the actual code that is generated, since that is machine dependent, but that the correct addressing can be performed.

```
let mkPair = proc (sTag, tTag : int →
                        proc (α, β → structure (fst : α ; snd : β)))
begin
    let fstOffset = 0
    let sndOffset = fstOffset + typeSize (sTag)
        ! typeSize yields the representation size for this type tag
    let structSize = sndOffset + typeSize (tTag)
        !This is required by the store allocator

    proc (first : α ; second : β → structure (fst : α ; snd : β))
        ! The structure size and its field addresses can be accessed
        ! in the static environment
    struct (fstOffset = first, sndOffset = second))
end
```

Fig. 15. Compiling Figure 14.

The technique is also flexible enough to allow any permutation of fields. The Napier88 system uses a canonical permutation to aid garbage collection. It places the pointers as one block at the start of the structure so that they can be easily found. The order may also be used as a canonical form to enable structural type equivalence across independently prepared program and data. The permuting of fields does, however, complicate the calculation of field offset addresses since the order of the fields is not known until run time.

It should also be noted that selecting a quantified type field still requires the conversion from concrete to polymorphic form and vice versa for assignment to such a field. This is, however, the same *convertToPoly* and *convertFromPoly* operations described above.

Although here we have dealt only with the addressing of fields of structures, the technique applies to all data structures.

3.4 Empirical Results and Implementation Tactics

The main advantage of the Napier88 technique is that only quantifier values are tagged. The tagging overhead is nonexistent for nonpolymorphic code. This is unlike uniform polymorphism, where all the values are regarded as polymorphic in their implementation. These are usually implemented as pointers giving a space overhead on every object and a dereferencing overhead on every access. In the Napier88 scheme, the overhead is only on conversion. van Vliet and Gladney have confirmed that tagging may be performed efficiently under such circumstances [41].

The second point to note is that for every call of a polymorphic procedure two procedure calls are made. The first is for the envelope procedure to set up the tag values, and the second is for the enclosed polymorphic form. The system is most efficient when users specialize the polymorphic procedure once to a particular type and then call the specialized type many times. Indeed, the normal usage of the Napier88 system is precisely that, in that polymorphic procedures are taken from the persistent store, specialized once, and used many times. Although in Napier88 the specialization has to be performed explicitly, type inferencing could be used to perform it automatically.

The third point on the efficiency of the technique concerns the cost of converting to and from the polymorphic forms. This depends on the efficiency of the tagging and the polymorphic representation. In Napier88 the tag encodes the concrete value size, and this can be used to aid the efficiency of the conversion. Some performance results are given in the Appendix.

By using a mixed implementation strategy, a number of improvements to the efficiency of the Napier88 scheme can be made. The first tactic is to compile away some polymorphic forms and substitute in-line code wherever possible. For example, the procedure call

```
id [int] (42)
```

need not even call the procedure.

Wherever appropriate, the textual polymorphism approach may be used. This is equivalent to partial application of the type parameters and trades the compact form of the polymorphic procedures against execution speed. Even in cases such as that in Figure 3, which has multiple quantifiers, it may be possible to compile away the variations when there are not too many calls. This is a decision for the compiler writer using static program analysis.

There are two situations where textual polymorphism will not help. The first is where there are many calls to a multiply quantified procedure. Since in Napier88 there are seven object forms, the number of polymorphic forms of a procedure with n quantifiers is seven to the power n. This gets large very quickly, and the method becomes inefficient.

The second case is where first-class polymorphic procedures are involved and textual polymorphism does not work. This is very common in Napier88, since these polymorphic procedures are often assigned to locations in the persistent store for later use. Indeed, this is the most common usage for such polymorphic forms. For that, the Napier88 procedural envelope technique is the appropriate implementation tactic.

The final tactic is to perform some static analysis to elide unnecessary conversions between forms. Figure 16 demonstrates a possible case. The calling sequence *that_id*(x) normally converts x to a nonpolymorphic form before the call and then converts it again inside the procedure. In this case, however, it is unnecessary since the procedure *that_id* cannot escape the scope of t and is always polymorphic.

In languages where the compiler can always determine the declared type of the procedure being called, it is possible to modify the Napier88 technique to

```
let this_id = proc [t] (x : t → t)
begin
    let that_id = proc (y : t → t) ; y
    that_id (x)
end
```

Fig. 16. Polymorphic optimization.

one that passes the quantifier tag as a parameter rather than holds it in the closure. In Napier88, since specializations such as

id [**int**]

are allowed and are substitutable with procedures of type **proc(int → int)**, then the compiler does not know statically whether to supply a type parameter or not.

Other languages, especially those that use type inferencing, do not allow specialization without call. In the Napier88 system, the greatest efficiency is achieved for a polymorphic procedure where it is specialized once and called many times. The cost of creating the envelope is amortized over many calls.

4. ABSTRACT DATA TYPES

Polymorphic expression occurs naturally in language constructs other than procedures. In Napier88, for example, there is a second mechanism for abstracting over type that yields abstract data types [36].

Abstract data types are characterized by having a state and an abstract interface through which the state may be manipulated. Many forms of abstract data types exist in different programming languages. The version that we discuss here is one where once the abstract data type is created then the only manner in which the internal state can be manipulated is by means of the interface. These are the existentially quantified types of Mitchell and Plotkin [33] and are in contrast to polymorphic procedures that have universally quantified types.

The power of existentially quantified types is the ability to create objects that have different component types but have the same abstract type. For example, an abstract data type interface may be created with an integer and a procedure to increment the integer by 1. Another may be created with a real number and a procedure to increment the real number by 1.0. Both of these have the same abstract interface to the outside world but, of course, have very different implementations. Since they have the same abstract interface (type), code may be written to manipulate them independently of their concrete type. This code constitutes polymorphic expression.

We now show how the tagged polymorphism implementation of polymorphic procedures in Napier88 may be used to implement these existentially quantified types. All the benefits of nonuniform object sizes and persistence of data objects are preserved.

First, we introduce the syntax of abstract data types in Napier88, in which the examples are written.

4.1 Napier88 Abstract Data Types

Napier88 provides a mechanism for abstracting over types that yields abstract data types. The declaration

type TEST **is abstype** [i] (value : i ; operation : **proc** (i → i))

declares the type *TEST* as abstract. The type quantifiers, enclosed in the square brackets, are called the witness type identifiers and are the types over which the abstraction takes place.

The abstract data type interface is declared between the round brackets. In the above example, the abstract data type interface has two elements, a field *value* with type *i* and a procedure *operation* with type **proc** (*i* → *i*).

To create an abstract data object, the type constructor identifier is used. For example,

let inc_int = **proc** (a : **int** → **int**) ; a + 1
let this = TEST [**int**] (3, inc_int)

declares the abstract data object *this* from the type definition *TEST*, the concrete (as opposed to abstract) witness type **int**, the integer 3, and procedure *inc_int*.

Once the abstract data object has been created, the user can never again tell how it was constructed. Thus, *this* has type **abstype**[i](*value* :i ; *operation* : **proc**(*i* → *i*)), and the user can never discover that the witness type is integer. The declaration

let that = TEST [**int**] (−42, inc_int)

creates another abstract data object, *that*. Although it is constructed using the same concrete witness type, this information is abstracted over. *this* and *that* have the same type, which is

abstype [i] (value : i ; operation : **proc** (i → i))

as does *also* below,

let inc_real = **proc** (b : **real** → **real**) ; b + 1.0
let also = TEST [**real**] (−41.99999, inc_real)

although it is created with a different witness type.

Since the internal representation of an abstract data object is hidden, it is inappropriate to mix operations from one with another. That is, the abstract data object is totally enclosed and may only be used with its own operations. A second requirement is that the type checking on the use of these objects is static. To achieve the above aims, the **use** clause is introduced to define a constant binding for the abstract data object. This constant binding can then be indexed to refer to the values in a manner that is statically checkable.

For example, in Figure 17, the procedure *manipulate* takes as a parameter an abstract data type of type *TEST*. Thus, it may be legally called with the values *this*, *that*, and *also*, which have type *TEST*. Inside the procedure, the

```
let manipulate = proc (this_one : TEST)
                    use this_one as X in
                    begin
                        X (value) := X (operation) (X (value) )
                    end
manipulate (this)
manipulate (that)
manipulate (also)
```

Fig. 17. Using abstract data types.

```
use this as X [B] in
begin
    let id = proc [t] (x : t → t) ; x
    let one := X (value)
    one := id [B] (one)
end
```

Fig. 18. Naming the witness type.

use clause binds the parameter to the identifier X, which is then used to access the fields of the abstract data type. For each call, the *value* field is passed to the *operation* procedure, and the result stored in the *value* field.

In the **use** clause, the witness types may be named for use. For example, procedures over these witness types may be written. This is shown in Figure 18. The witness type is renamed B, which is then allowed to be used as a type identifier within the **use** clause.

The utility of existential types as a database viewing mechanism is explored in [15]. We now demonstrate how the technique for implementing polymorphic procedures may be used to implement the existentially quantified type objects described above in a nonuniform store.

4.2 Implementing Abstract Data Types

Central to the understanding of how abstract data types may be implemented is the realization that the **use** clause forms a polymorphic block. When compiling the clause, the compiler does not know the witness types of any of the abstract data types and must compile code that will operate for all types. This is analogous to polymorphic procedures, and a similar technique may be used to solve the problem.

The polymorphic values within the **use** clause are objects of the witness type itself (cf. quantifier parameters). For all other values, the type and, therefore, the representation are known. The clause can be compiled in the same manner as a polymorphic procedure by referring to a witness tag that is

held within the closure of the block. This witness tag can be found from the concrete value on dynamic entry to the clause.

Since the creation of abstract data types is separate from their use and many different ADTs may use the same **use** clause, a mechanism for initializing the witness tag in the **use** clause to the particular witness type is required. This is done by storing the witness tag with the abstract data object and retrieving it before entry to the **use** clause.

The abstract data type itself may be implemented as a structure of isomorphic shape plus fields of the witness tags. For example, the creation clause

```
let this = TEST [int] (3, inc_int)
```

can be compiled to

```
let this = struct (iTagOffset := intTag, valueOffset := 3, operationOffset :=
  inc_int)
```

In the above, the abstract data type has two fields: *value* and *operation*. The structure representing it has the same fields plus a tag field for every witness type.

The addresses of the fields can be calculated from the specialization types. They are either known statically or are quantifier types, in which case the code for creating a structure within a polymorphic procedure is generated, or they are witness types of outer scope abstract data types. This last case is the same as the second since, for the concrete witness type to be another witness type, the creation must be within a **use** clause, for which a tag is available. The **use** clause can now calculate the addresses of the fields of the structures from the tag. Thus,

```
use this as X in
begin
  X (value) := X (operation) (X (value))
end
```

compiles to the code in Figure 19.

The calculation of the offsets for the fields in the abstract data type is exactly the same as the one performed for calculating the addresses in a structure, which have fields of a quantifier type, within a polymorphic procedure.

There still remains one difficulty with this method. In order to calculate the field offset addresses on entry to the **use** clause, the system must know all the tags. To find a witness tag, the system must know its field offset address. In the above, the witness tags are stored in fixed locations, which are the same for all abstract data objects of that type. Thus, for any **use** clause, the tags can be found and the addresses calculated.

This restriction can be lifted by storing the fields of the witness types in the structure in a fixed form, for example, in their polymorphic representation. Where all the other fields are of a fixed size, which is the case except where they have quantifier or other witness types, the addresses of the fields can be calculated statically saving the dynamic calculation. Thus, this also yields an addressing optimization.

```
let X = this
let iTagOffset = 0
let iTag = X (iTagOffset)
let valueOffset = iTagOffset + typeSize (int)
let operationOffset = valueOffset + typeSize (iTag)
begin
        X (valueOffset) := X (operationOffset) (X (valueOffset))
end
```

Fig. 19. Code generation for a use clause.

The optimization is possible since the abstract data type may only be used within a **use** clause and since its concrete type may never be accessed by another means. This is in direct contrast to quantifier fields in structures that have to assume their concrete type without the polymorphic procedure. It should be noted that, for the same reason, only witness fields of the particular abstract data type need be stored in the data structure in the polymorphic form.

The technique also works for any permutation of fields within the abstract data type. This is done in Napier88 to aid garbage collection, but also has the disadvantage of slightly complicating the field offset address calculations.

Finally, the method described works for any combination of polymorphic procedures and abstract data types, both of which are first-class data objects.

5. CONCLUSIONS

We have presented a technique for implementing polymorphism, which we have used to support the language Napier88. It is a variant of tagged polymorphism and can be used to implement parametric and built-in ad hoc polymorphism. Combined with the work of Wadler and Blott [42], it can be used to implement user-defined ad hoc polymorphism, although, as yet, we have not done this. We have also used this method to implement the existentially quantified types of Mitchell and Plotkin [33]. Finally, we have demonstrated elsewhere that, since the method can implement ad hoc polymorphism in a nonuniform store, it can also implement bounded universal quantification, bounded existential quantification, and inclusion polymorphism [14].

The technique may also be used to implement all data types that exhibit universal quantification. This is a direct consequence of the fact that all values in the λ-calculus are functions and that it is sufficient to provide a technique that will implement polymorphic functions such as the one described in this paper.

Of our original design goals, we can

—implement all forms of polymorphism,

—use a conventional architecture,

—support nonuniform data objects,
—support first-class polymorphic forms,
—support the persistence of polymorphic forms,
—only require extra space and execute additional code in polymorphic contexts, and
—generate compact and fast code.

This last goal is achieved by varying the tactics to suit the circumstances.

APPENDIX

The following timings were taken on a SUN 4/360 running the Napier88 system. The purpose of the first set of tests was to evaluate the cost of the polymorphic procedure call. The three procedures

```
let dummy = proc ( ) ; { }
let polyDummy = proc [t] ( ) ; { }
let specializedPolyDummy = polyDummy [int]
```

were called 100,000 times; the average time of the call is detailed in Table I.

The time to call the envelope procedure is 79 μs, compared with 24 μs for a normal procedure call. This may be accounted for by the fact that there are two procedure calls, the first of which takes the tag as a parameter and produces a result that is the closure of the polymorphic procedure. However, there is no overhead for calling the polymorphic procedure once it has been specialized.

The second test was devised to find the cost of conversion to the polymorphic form. The following procedures were used:

```
let Dummy = proc (x : int) ; { }
let polyDummy = proc [t] (x : t) ; { }
let specializedPolyDummy = polyDummy [int]
```

They were again called 100,000 times. The times in Table I were subtracted from the average time to yield the parameter passing times in Table II. The extra cost of calling a procedure with a parameter of a quantified type is the cost of performing the *convertToPoly* instruction. These times were verified using a polymorphic procedure with two parameters.

The third test evaluated the cost of the conversion from the polymorphic form. The following procedures were used, as before:

```
let intIdentity = proc (x : int → int) ; x
let identity = proc [t] (x : t → t) ; x
let specializedIntIdentity = identity [int]
```

The results are given in Table III.

The extra cost of returning from the polymorphic procedures is the cost of performing the *convertFromPoly* instruction. It should be noted that it takes 11 μs to execute both the *convertToPoly* and *convertFromPoly* instructions.

ACKNOWLEDGMENTS

We acknowledge the thorough proofreading and suggestions for the improvement of this paper given by Malcolm Atkinson. Quintin Cutts and Graham

Table I. Average Time to Call a Procedure

Procedure	Time
dummy	24 μs
polyDummy	79 μs
specializedPolyDummy	24 μs

Table II. Average Time to Pass a Parameter

Procedure	Time
dummy	2 μs
polyDummy	13 μs
specializedPolyDummy	13 μs

Table III. Average Time to Return a Value

Procedure	Time
intIdentity	10 μs
identity	21 μs
specializedIntIdentity	21 μs

Kirby are also thanked for their proofreading and suggestions. Finally, we know that David Stemple is correctly to be credited with the term *text editor polymorphism*, which we have carefully avoided referencing until now.

REFERENCES

1. Albano, A., Cardelli, L., and Orsini, R. Galileo: A strongly typed conceptual language. *ACM Trans. Database Syst. 10*, 2 (June 1985), 230-260.
2. Atkinson, M. P., and Morrison, R. Procedures as persistent data objects. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 539-559.
3. Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P., and Morrison, R. An approach to persistent programming. *Comput. J. 26*, 4 (Nov. 1983), 360-365.
4. Birtwistle, G. M., Dahl, O. J., Myrhaug, B., and Nygaard, K. *SIMULA BEGIN*. Auerbach, Pennsauken, N.J., 1973.
5. Boehm, B. W. Understanding and controlling software costs. In *Tenth IFIP World Congress* (Dublin, Ireland, Sept. 1-5, 1986). North-Holland, Amsterdam, 1986, pp. 703-714.
6. Brown, A. L., Carrick, R., Connor, R. C. H., Dearle, A., and Morrison, R. The persistent abstract machine. Res. Rep. PPRR-59-88, Dept. of Computational Science, Univ. of St. Andrews, Scotland, 1988. (Also published by the Univ. of Glasgow, Scotland.)
7. Burstall, R., and Lampson, B. A kernal language for abstract data types and modules. In *Proceedings of the International Symposium on the Semantics of Data Types* (Sophia-Antipolis, France). Lecture Notes in Computer Science, vol. 173. Springer-Verlag, New York, 1984.
8. Burstall, R., McQueen, D., and Sanella, D. Hope: An experimental applicative language. In *ACM Lisp Conference* (Stanford, Calif., Aug. 25-27, 1980), ACM, New York, 1980, pp. 136-143.
9. Cardelli, L. The functional abstract machine. *Polymorphism Newsl. 1*, 1 (Jan. 1983).
10. Cardelli, L. Basic polymorphic type checking. *Polymorphism Newsl. 2*, 1 (Jan. 1984). (Also Tech. Rep. 119, AT&T Bell Labs.)

11. CARDELLI, L. A semantics of multiple inheritance. In *Semantics of Data Type Lecture Notes in Computer Science, 173*. Springer-Verlag, New York, 1984, pp. 51-67.
12. CARDELLI, L., AND WEGNER, P. On understanding types, data abstraction and polymorphism. *ACM Comput. Surv. 17*, 4 (Dec. 1985), 471-523.
13. CONNOR, R. C. H. The Napier type checking module. Res. Rep. PPRR-58-88, Dept. of Computational Science, Univ. of St. Andrews, Scotland, 1988. (Also published by Univ. of Glasgow, Scotland.)
14. CONNOR, R. C. H., DEARLE, A., MORRISON, R., AND BROWN, A. L. An object addressing mechanism for statically typed languages with multiple inheritance. In *OOPSLA89* (New Orleans, La.) *ACM SIGPLAN Not. 24*, 10 (Oct. 1989), 279-286.
15. CONNOR, R. C. H., DEARLE, A., MORRISON, R., AND BROWN, A L. Existentially quantified types as a database viewing mechanism. In *Advances in Database Technology—EDBT90* (Venice, 1990). Lecture Notes in Computer Science, vol. 416. Springer-Verlag, New York, 1990, pp. 301-315.
16. CONNOR, R. C. H., BROWN, A. L., CARRICK, R., DEARLE, A , AND MORRISON, R. The persistent abstract machine In *Third International Workshop on Persistent Object Systems* (Newcastle, N.S.W., Australia, Jan. 10-13, 1989). Persistent Object Systems. Springer-Verlag, New York, 1989, pp. 353-366.
17. DAMAS, L., AND MILNER, R. Principal type schemes for functional programs In *Ninth ACM Symposium on Principles of Programming Languages* (Albuquerque, N.M., Jan. 25-27, 1982). ACM, New York, 1982, pp. 207-212.
18. DEARLE, A., AND BROWN, A. L. Safe browsing in a strongly typed persistent environment. *Comput. J. 31*, 6 (Dec. 1988), 540-545.
19. DEMERS, A., AND DONAHUE, J. Revised report on Russell. Tech. Rep. TR79-389, Dept. of Computer Science, Cornell Univ., Ithaca, N Y , 1979.
20. FAIRBAIRN, J. Ponder and its type system Tech. Rep. 31, Dept. of Computer Science, Univ. of Cambridge, U.K., Nov. 1982.
21. FAIRBAIRN, J. A new type checker for a functional language. In *Data Types and Persistence*, Topics in Information Systems Series. M. P. Atkinson, O. P. Buneman, and R. Morrison, Eds. Springer-Verlag, New York, 1988, pp 69-88.
22. GOLDBERG, A., AND ROBSON, D. *SMALLTALK-80: The Language and Its Implementation*. Addison-Wesley, London, 1983
23 HAMMER, M , AND MCLEOD, D Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept. 1981), 351-386.
24 HAUCK, E. A., AND DENT, B. A. Burroughs B6500/6700 stack mechanism. *AFIPS SJCC 32* (Atlantic City, N.J., Apr. 30-May 2, 1968), pp. 245-252.
25. HUDAK, P., WADLER, P., ARVIND, BOUTEL, B., FAIRBAIN, J., FASEL, J., HAMMOND, K., HUGHES, J., JOHNSON, T., KIEBURTZ, D., NIKHIL, R., PEYTON JONES, S., REEVES, M , WISE, D., AND YOUNG, J. Report on the functional programming language Haskell. Dept. of Computer Science, Glasgow Univ., Scotland, Apr. 1990. (Also published by Yale Univ., New Haven, Conn.)
26. ICHBIAH ET AL. *The Programming Language Ada Reference Manual*. Lecture Notes in Computer Science, vol. 155. Springer-Verlag, New York, 1983.
27. JOHNSTON, J. B. The contour model of block structure processes. *ACM SIGPLAN Not. 6*, 2 (Feb. 1971), 56-82.
28. KEAS, S. *Parametric Overloading in Polymorphic Programming Languages*. Lecture Notes in Computer Science, vol. 300. Springer-Verlag, New York, 1988, pp. 131-144.
29. LISKOV, B. H. *CLU Reference Manual*. Lecture Notes in Computer Science, vol. 114. Springer-Verlag, New York, 1981.
30. MATTHEWS, D. C. J. Poly manual. Tech. Rep. 65, Dept. of Computer Science, Univ. of Cambridge, U.K., 1985.
31. MILNER, R. A proposal for standard ML. Tech. Rep. CSR-157-83, Dept. of Computer Science, Univ. of Edinburgh, Scotland, 1983.
32. MILNER, R. A theory of type polymorphism in programming. *J. Comput Syst. Sci. 17*, (1978), 348-375.

33. MITCHELL, J. C., AND PLOTKIN, G. D. Abstract types have existential type. In *Twelfth ACM Symposium on Principles of Programming Languages* (New Orleans, La., Jan. 14-16, 1985). ACM, New York, 1985, pp. 37-51.
34. MORRISON, R., BROWN, A. L., CONNOR, R. C. H., AND DEARLE, A. The Napier88 Reference Manual. Res. Rep. PPRR-77-89, Dept. of Computational Science, Univ. of St. Andrews, Scotland, 1989. (Also published by the Univ. of Glasgow, Scotland.)
35. MORRISON, R., BROWN, A. L., CARRICK, R., CONNOR, R. C. H., DEARLE, A., AND ATKINSON, M. P. Polymorphism, persistence and software reuse in a strongly typed object-oriented environment. *Softw. Eng. J.* (Nov. 1987), 199-204.
36. MORRISON, R., BROWN, A. L., CARRICK, R., CONNOR, R. C. H., DEARLE, A., AND ATKINSON, M. P. The Napier type system. In *Third International Workshop on Persistent Object Systems* (Newcastle, N.S.W., Australia, Jan. 1989). Persistent Object Systems. Springer-Verlag, New York, 1989, pp. 3-18.
37. STEMPLE, D., FEGARAS, L., SHEARD, T., AND SOCORRO, A. Exceeding the limits of polymorphism in database programming languages. In *Advances in Database Technology—EDBT90* (Venice, 1990). Lecture Notes in Computer Science, vol. 416. Springer-Verlag, New York, 1990, pp. 269-285.
38. STRACHEY, C. *Fundamental Concepts in Programming Languages.* Oxford University Press, Oxford, U.K., 1967.
39. TURNER, D. A. SASL language manual. Rep. CS/79/3, Dept. of Computational Science, Univ. of St. Andrews, Scotland, 1979.
40. TURNER, D. A. *Miranda System Manual.* Research Software, Ltd., Canterbury, England, 1987.
41. VAN VLIET, J. C., AND GLADNEY, H. M. An evaluation of tagging. *Softw. Pract. Exper. 15*, 9 (Sept. 1985), 823-827.
42. WADLER, P., AND BLOTT, S. How to make ad hoc polymorphism less ad hoc. In *Sixteenth ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan. 1989). ACM, New York, 1989, pp. 60-76.

Received November 1989; revised November 1990; accepted January 1991