Modula-3 as a Database Programming Language

Ryan Stansifer
Department of Computer Sciences
University of North Texas
Denton, Texas 76203–3886
Internet: stansifer@cs.unt.edu
Telephone: (817) 565–2811

September 6, 1994

1 Introduction

In the paper *Types and Persistence in Database Programming Languages*, Atkinson and Buneman compare several approaches to typical database programming problems. In addition to the more traditional database languages like CODASYL, Pascal/R, and Galileo, several general-purpose programming languages are included in their comparison. Among these are Pascal, ML, and Amber. This raises the question: how suitable are general-purpose programming languages as database languages?

Although is impossible to answer the question definitively, we take up the database problems posed by Atkinson and Buneman and solve them with the general-purpose programming language Modula-3. Modula-3 is a recent programming language ideally suited among the general-purpose languages to add to the comparison. To be sure, natural expression of algorithmic content and a naturally useful type system make programming languages better in general. We argue that this case study adds evidence to the claim that these improvements make general-purpose languages more appealing for specialized applications (even diverse applications).

Database programming languages must provide some advantages or there would not be so many of them. The obvious advantages seem to be specialized functionality, ease of expression, and performance. In this paper we test these beliefs by taking a general purpose language, Modula-3, and using it to solve some challenges in database programming.

1.1 Database programming languages

We are concerned with language design issues—the natural expression of algorithmic ideas—not implementation issues. Implementation issues are, of course, a critical part of database languages which are typically concerned with quick access to large amounts of data.

In large part, general purpose programming languages handle implementation well. Procedural and data abstraction mechanisms hide the implementation details. Most languages provide modules that allow the programmer to specify a data structure, a dictionary, say. The implementation of the operations for creating dictionaries, searching for entries, and updating entries can be hidden from the casual programmer in libraries, just like mathematical operations. Different implementations for different circumstances are often available in standard libraries In fact, the casual programmer may have more difficulty assessing the efficiency of the implementation than in writing the program. Of course, the benefits from libraries increase when programming is concentrated in fewer programming languages, since more effort is put into reuse and less into starting from scratch.

1.2 Modula-3

Modula-3 is a strongly-typed systems language with a Pascal-like core, modules, and objects. It is an imperative, block structured language; and it has exception handling, modules, and generics. Despite its name, Modula-3 is not a revision of Modula-2 and was actually designed from scratch. Modula-3 shares with Modula-2 a Pascal-like core and modules with interface and implementation files.

Modula-3 was designed jointly by Digital Equipment Corporation's Systems Research Center and the Olivetti Research Center. The designers were Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow and Greg Nelson. The fifty-page reference manual (a fifty-page reference manual was a design goal) has been published [10, chapter 2]. Some other sources about the language have already appeared [3, 2] and a high-level X Window System toolkit is being developed in Modula-3 [6, 7].

Figure 1 shows a simple Modula-3 program, the same TPK program used by Knuth and Pardo [5] to compare many early programming languages. The TPK program does not do anything particularly useful, but does illustrate a few aspects of programming. The program reads in 11 real numbers into an array and, in reverse order, applies a function to each value. The result is then printed if it is not too large.

Notice that the square-root operation is not an integral part of Modula-3, but rather belongs to a library of mathematical subroutines (actually just the standard math library of the C programming language). This library is accessed through a Modula-3 interface module called `Math` which proves the names and types of all the procedure and functions so that type-safe linking is assured.

Input/output is handled by streams called "readers" and "writers" in the interfaces `Rd` and `Wr`. Strings can be printed by the procedure `PutText`. The `&` operator concatenates two strings.

Lots of other modules are provided with the Modula-3 interface including sets, lists, hash tables and other data structures. One important module is the `Pkl` module. The `Pkl` module provides type-safe binary storage for linked data structures. In Modula-3 persistent objects are obtained by *pickling* them. The data resides in a file with all its type-information and pointers intact. Because Modula-3 uses structural type equivalence, another program can read the pickled data as long as it knows the structure of the data. Another important module is the `Thread` module. The `Thread` module provides control of concurrent programs.

2 The tasks

In the paper *Types and Persistence in Database Programming Languages* [1], Atkinson and Buneman compare several approaches to typical database programming problems. They pose four tasks. We solve these tasks using Modula-3.

The first task is to describe the database.

> Task 1: *Describe the database.*

This is necessary so that the basic data structure representing the organization of the database is revealed.

The second task is a simple query.

> Task 2: *Print the name, cost, and mass of all parts that cost more than $100.*

Task 3 is an example of something that is impossible to do within relational algebra, hence impossible in most relational query languages.

```
MODULE TPK EXPORTS Main;

IMPORT Math;                         (* Interface to C math library.   *)
IMPORT Wr, Rd, Stdio, Fmt, Scan; (* Standard library modules.      *)

PROCEDURE f (x: LONGREAL): LONGREAL =
BEGIN
    RETURN (Math.sqrt (ABS (x)) + 5.0D0*Math.pow(x,3.0D0));
END f;

VAR
    A : ARRAY [0 .. 10] OF LONGREAL;

BEGIN
    FOR i := FIRST(A) TO LAST(A) DO
        A[i] := Scan.LongReal (Rd.GetLine (Stdio.stdin));
    END;

    FOR i := LAST(A) TO FIRST(A) BY -1 DO
        WITH
           y = f (A[i]),  t = Fmt.Int (i) & "  "
        DO
           IF (400.0D0 >= y) THEN
               Wr.PutText (Stdio.stdout, t & Fmt.LongReal (y) & "\n");
           ELSE
               Wr.PutText (Stdio.stdout, t & "TOO LARGE\n");
           END;
        END;
    END;
END TPK.
```

Figure 1: An example of a Modula-3 program

Task 3: *Print the total mass and total cost of a composite part.*

Examples like this motivate designers of database languages to include constructs to express all algorithmic possibilities. Thereby making them more like general-purpose programming languages.

The fourth task is update the database. This useful in exposing how well the database is organized and how easy it is to manipulate the data structures.

Task 4: *Record a new manufacturing step in the database, that is, how a new composite part is manufactured from subparts.*

3 The programs

3.1 Description of the database

The parts database consists of parts and suppliers. For the purposes of this database, parts are either base or composite. Individual base parts are supplied by one or more suppliers. Composite parts are manufactured out of other parts. The code is shown in figure 2. Notice that `BasePart` and `CompositePart` are subtypes of `Part`. This means that values of these types can be used where every a `Part` could be used.

The suppliers of a part really form a set; i.e., it makes no sense for the same supplier to appear more than once in the collection of suppliers. The same is true for the collection of component parts. Although we have chosen to represent these concepts as lists, using sets would not have been harder. A prefined module for sets is distributed with Modula-3 and it has an interface that is similar to that of lists.

Not only have we used lists, but we have used lists of pointers to anything, so-called `REFANY` lists. To proceed in a type-safe manner we would use lists whose element types were the appropriate ones. A generic list module is available in the standard distribution of Modula-3. To use a generic, an interface and a module must be instantiated from the generics ones. However this is impossible in this case as it would introduce a circular dependency between modules.

The problem is that to instantiate the generic list interface we need an interface defining the type `Use`. To define `Use` we need the type `CompositePart`. But to define `CompositePart` we need the type for lists of `Use`. Mutually recursive type definitions are allowed in Modula-3, but circular dependencies of one interface on another is not. The requirement that generics be used limits the flexibility of mutually recursive type definitions.

This demonstrates a serious weakness of using generics to accomplish universal polymorphism vis-à-vi more flexible languages like ML [9].

```
datatype
    CompositePart = CompositePart of {name: string, uses: Use list}
and
    Use = Use of {component: CompositePart, quantity: int}
```

```
INTERFACE Schema;   (* Parts database of Atkinson and Buneman *)

IMPORT RefanyList;

TYPE
    Dollars = INTEGER;
    PosInt  = [1..LAST(INTEGER)];
    Grams   = INTEGER;

    Part = OBJECT
        name:    TEXT;
        used_in: RefanyList.T;   (* Set of CompositePart *)
    END;

    BasePart = Part OBJECT
        cost:        Dollars;
        mass:        Grams;
        supplied_by: RefanyList.T; (* Set of Supplier *)
    END;

    CompositePart = Part OBJECT
        assembly_cost:  Dollars;
        mass_increment: Grams;
        uses:           RefanyList.T; (* Set of Use *)
    END;

    Use = OBJECT
        component: Part;
        assembly:  CompositePart;
        quantity:  PosInt;
    END;

    Supplier = TEXT;

    (* Database is a list of parts and a list of suppliers. *)
    DB = RECORD parts: RefanyList.T; suppliers: RefanyList.T END;

    PROCEDURE GetDB (READONLY file_name: TEXT): REF DB;
    PROCEDURE PutDB (READONLY file_name: TEXT; READONLY db: REF DB);

END Schema.
```

Figure 2: Interface file for parts database

```
MODULE Expensive EXPORTS Main;

IMPORT RefanyList;          (* Module for list of pointers to anythings *)
IMPORT Wr, Stdio, Fmt;      (* Standard library modules *)
IMPORT Schema;              (* Module for access to parts database *)

VAR
    db: REF Schema.DB;
    list: RefanyList.T;

BEGIN
    db := Schema.GetDB ("parts.pkl");
    list := db.parts;
    WHILE list # NIL DO
        (* The order of the following cases is important.  Subtypes must
           appear after supertypes so that the "best" type is found. *)
        TYPECASE list.first OF
        | Schema.BasePart (base) =>
            IF (base.cost >= 100) THEN
                Wr.PutText (Stdio.stdout,   "name=" & base.name);
                Wr.PutText (Stdio.stdout, ", cost=" & Fmt.Int(base.cost));
                Wr.PutText (Stdio.stdout, ", mass=" & Fmt.Int(base.mass) & "\n");
            END;
        | Schema.CompositePart (comp) =>          (* Do nothing *)
        (* Every element of "list" should be either base or composite part *)
        | Schema.Part (part) => <* ASSERT FALSE *>
        ELSE <* ASSERT FALSE *>
        END;
        list := list.tail;
    END;
END Expensive.
```

Figure 3: Finding all expensive parts

3.2 Finding expensive parts

Finding base parts in the database is representative of the simplest kind of database querys. The code for the query ought to be extremely natural. The program in Modula-3 is shown in figure 3. It is the code one would expect for a general purpose programming language. It is somewhat verbose compared to database languages, but fairly clear.

The could perhaps be improved by using the "apply" function in the list interface. The "apply" function takes a function as an argument and applies it to everything in the database. The only special purpose code necessary would be the case analysis; this approaches the simplicity of database languages. However, passing procedures in imperative languages is still alarming to programmers unacquainted with functional programming techniques. Incidentally, in Modula-3 only those procedures declared at the top-level of a module many be passed around as an argument, so the traditional run-time stack implementation is possible for Modula-3.

Notice that the program `Expensive` uses dynamic dispatch. The type of a part is explicitly consulted in order to determine which action to perform. The **`TYPECASE`** construct makes this possible in Modula-3. However, this is not particularly good programming practice [4]. In an object-oriented programming language (like Modula-3) it ought to be possible to attach a method to each kind of part and avoid the explicit **`TYPECASE`** statement. This is indeed possible in Modula-3 for any given predefined problem. But this does not seem to be in the spirit of the task. Every extemporaneous query to the database, if it will distinguish between subtypes, must provide actions (code) for the distinct cases. As the query will most likely never be used again, there is little point in encorporating all such queries as integral parts of the database.

3.3 Computing cost and mass

Computing the cost and mass of composite objects is an example of a very challenging query going to the core of the database's organization. The notion is quite simple, however. The cost and mass of a composite part depends on the cost and mass of its subparts. As subparts may in turn have subparts, this notion is recursive. The code is sketched in figure 4. The code contains the familiar loop through the database. When the part named "bicycle" is found its cost and mass are found by calling the subprocedure `CostMass`. This subprocedure and the two auxiliary procedures `CostMassBase` and `CostMassComp` are shown in figure 5.

The program treats the database as if it were organized like a tree. Each composite part is the root of tree containing subparts and their subparts until the leafs of base parts are reached. Because parts may be shared a graph search may be more efficient. One way to accomplish this kind a search without modifying the structure of the database, is to use a memo function. This can be done in Modula-3 as is explained in appendix A.

3.4 Updating the database

The code for the fourth task is sketched in figure 6. The unimportant subprocedure `GetInteger` is omitted, but otherwise the entire interactive part of the program is given. One part is added to the database, and the user is prompted for the part by name. The user also provides all the subcomponents by name. These parts must already exist in the data base. The subprocedure `Find` loops through the database to find a part. Its body has been omited, but its implementation is unremarkable.

4 Conclusion

Modern database languages appear more and more like general-purpose programming languages. Now they include highly structured data and universal control structures. This raises important questions. What is a programming language? What is the role of special purpose languages? Do we need both types?

Modula-3 fares well with regard to the primary points raised in [1] about the design of database programming languages.

```
MODULE CostMass EXPORTS Main;  (* compute cost and mass of a composite part *)

IMPORT RefanyList
IMPORT Text, Wr, Stdio, Fmt;
IMPORT Schema;

PROCEDURE CostMassBase (* ... *)
PROCEDURE CostMassComp (* ... *)
PROCEDURE CostMass     (* ... *)

VAR
    db: REF Schema.DB;      (* The entire database  *)
    list: RefanyList.T;     (* list of parts really *)

BEGIN
    db := Schema.GetDB ("parts.pkl");
    list := db.parts;
    WHILE list # NIL DO     (* loop through all the parts *)
        TYPECASE list.first OF
        | Schema.Part (part)  =>
            IF Text.Equal (part.name, "bicycle") THEN
                VAR
                    cost: Schema.Dollars;  mass: Schema.Grams;
                BEGIN
                    CostMass (part, cost, mass);
                    Wr.PutText (Stdio.stdout,   "name=" & part.name);
                    Wr.PutText (Stdio.stdout, ", cost=" & Fmt.Int (cost));
                    Wr.PutText (Stdio.stdout, ", mass=" & Fmt.Int (mass) & "\n");
                END;
            END;
        ELSE <* ASSERT FALSE *>
        END;
        list := list.tail;
    END;
END CostMass.
```

Figure 4: Computing cost and mass of a composite part

```
PROCEDURE CostMassBase (
    READONLY base: Schema.BasePart;
    VAR cost: Schema.Dollars;
    VAR mass: Schema.Grams
) =
    BEGIN
        cost := base.cost;
        mass := base.mass;
    END CostMassBase;

PROCEDURE CostMassComp (
    READONLY comp: Schema.CompositePart;
    VAR cost: Schema.Dollars;
    VAR mass: Schema.Grams
) =
    VAR
        list: RefanyList.T;
        subtotal_cost: Schema.Dollars;
        subtotal_mass: Schema.Grams;
    BEGIN
        cost := comp.assembly_cost;
        mass := comp.mass_increment;
        list := comp.uses;
        WHILE list # NIL DO
            WITH
                use = NARROW (list.first, Schema.Use)
            DO
                CostMass (use.component, subtotal_cost, subtotal_mass);
                INC(cost, use.quantity * subtotal_cost);
                INC(mass, use.quantity * subtotal_mass);
            END;
            list := list.tail
        END;
    END CostMassComp;

(* Compute cost and mass of either base or composite parts *)
PROCEDURE CostMass (
    READONLY part: Schema.Part;
    VAR cost: Schema.Dollars;
    VAR mass: Schema.Grams
) =
    BEGIN
        TYPECASE part OF
        | Schema.BasePart (base)      => CostMassBase (base, cost, mass);
        | Schema.CompositePart (comp) => CostMassComp (comp, cost, mass);
        | Schema.Part (part)          => <* ASSERT FALSE *>
        END;
    END CostMass;
```

Figure 5: Computing cost and mass of a composite part (subroutines)

```
MODULE Update EXPORTS Main;      (* Add 1 composite part to the database *)
IMPORT RefanyList, Text, Rd, Wr, Stdio, Fmt, Scan, Schema;
PROCEDURE GetInteger (msg: TEXT): INTEGER =  (* Omitted *)
PROCEDURE Find (list: RefanyList.T; name: Text.T): Schema.Part = (* Omitted *)
VAR
    db:   REF Schema.DB;              (* The entire database *)
    name: Text.T;                     (* Name of part, supplied by user    *)
    cost: Schema.Dollars;             (* Cost of assembly, supplied by user *)
    mass: Schema.Grams;               (* Mass increment, supplied by user *)
    part: Schema.CompositePart;       (* Local variable to hold new part *)
    uses_list: RefanyList.T:=NIL; comp: Schema.Part;  quan: INTEGER;  use: Schema.Use;
BEGIN
    db := Schema.GetDB ("parts.pkl");
    Wr.PutText (Stdio.stdout, "Enter name of new composite part -- ");
    Wr.Flush (Stdio.stdout);
    name := Rd.GetLine (Stdio.stdin);
    <* ASSERT Find (db.parts, name) = NIL *> (* Names are supposed to be unique *)
    cost := GetInteger ("assembly cost");
    mass := GetInteger ("mass increment");
    part := NEW(Schema.CompositePart,name:=name,assembly_cost:=cost,mass_increment:=mass);
    Wr.PutText (Stdio.stdout, "Enter names of component parts, one per line.\n");
    Wr.PutText (Stdio.stdout, "An empty line terminates the list.\n");
    Wr.PutText (Stdio.stdout, "Component parts must already exist in the db.\n");
    LOOP
        Wr.PutText (Stdio.stdout, "Enter name of existing component part -- ");
        Wr.Flush (Stdio.stdout);
        name := Rd.GetLine (Stdio.stdin);
        IF (Text.Empty (name)) THEN EXIT END;
        quan := GetInteger ("quantity of component part");
        comp := Find (db.parts, name);
        IF (comp # NIL) THEN
            use := NEW(Schema.Use,component:=comp,assembly:=part,quantity:=quan);
            RefanyList.Push (uses_list, use);
            RefanyList.Push (comp.used_in, part);
        ELSE
            Wr.PutText (Stdio.stdout, "Component part "&name&" doesn't exist!\n");
        END;
    END;
    part.uses := uses_list;
    RefanyList.Push (db.parts, part);  (* Destructive update *)
    Schema.PutDB ("parts.pkl", db);    (* Write out the modified database *)
END Update.
```

Figure 6: Updating the database

- There should be a uniform language for development of database programs. Clearly this is a big advantage in using existing general-purpose programming languages for database applications. There would be no need to learn two languages, one for describing algorithms and the other for database applications.

- Persistence should be independent of type. In Modula-3 the `Pkl` interface works for all types, even pointers.

- Abstract data types should be available to represent the regularity of the database. On the whole, modern general-purpose languages do this well using some construct for abstract data types. Modula-3 may do this better than many programming languages.

- Polymorphic indexing is required for efficient implementation. While this is not built into the programming language Modula-3, standard library routines exist for common structures, like hash tables for instance. The point is that the language permits, even encourages, construction of specialized modules for all the needs and trade-offs of database programming.

- Procedures should be typed, persistent objects. They are in Modula-3.

- They type system should have some form of inheritance. It does in Modula-3.

- Incremental or dynamic binding should be permitted. The `REFANY` type in Modula-3 is a good approximation to this.

- As much type checking should be done by the compiler as possible. Modula-3 is strongly typed.

Modula-3 was not as impressive as some of the languages in the Atkinson and Buneman study in eliminating pointers and taking advantage of polymorphism. The clumsy generic mechanism is not as flexible as the one in languages with polymorphism integrated with the type system. But otherwise seems quite adequate for the database tasks posed.

These issues cover just a part of the range of issues connected with database design. Some topics, like locking and concurrency control, have been left out entirely. But the point is that it does not matter. A general-purpose language with the right constructs can be used to create specialized functionality in as natural a manor as a special-purpose programming language.

Pluralitas non est ponenda sine necessitate.

William Occam, *Quodliberta*, ca. 1324

References

[1] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computng Surveys*, 19(2):105–190, June 1987.

[2] S. P. Harbison. Modula-3. *Byte*, 15(12):385, October 1990.

[3] S. P. Harbison. *Modula-3*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[4] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[5] D. E. Knuth and L. T. Pardo. The early development of programming languages. In *Encyclopedia of Computer Science and Technology*, pages 419–96, New York, 1977. Marcel Dekker.

[6] M. S. Manasse and G. Nelson. Trestle reference manual. SRC Report 68, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, December 1991.

[7] M. S. Manasse and G. Nelson. Trestle tutorial. SRC Report 69, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1992.

[8] D. Michie. 'Memo' functions and machine learning. *Nature*, 218(5136):19–22, April 1968.

[9] R. Milner, M. Tofte, and R. W. Harper, Jr. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[10] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[11] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, November 1990.

A Memoizing

Task 3 is to compute the cost and mass of a given composite part. The way the database is organized it is possible that different composite parts can share the same parts. In other words, the structure may be a graph and not a tree. For example, a portion of the parts database may contain information about manufacturing a bicycle.

```
bicycle
    handlebar (1)
        grip (2)
    front brake (1)
        short break cable (1)
        break shoe (2)
            rubber pad (1)
            metal assembly (1)
            nut (1)
            washer (1)
```

rear break (1)
  long break cable (1)
  break shoe—see front brake (2)
pedal (2)
  glass reflector (2)
    nut (2)
    bolt (2)

The break shoes on the bicycle are the same whether they are on the front or the back. The algorithm in figure 4 will compute the cost and mass of a break shoe twice, once for the front break and again for the back break.

This extra computation can be spared if some way can be devised to remember that the cost and mass of a break shoe has already been calculated. This trick is known as *memoization* [8].

The code for computing the cost and mass using memoization is sketched in figure 7. For the most part it is identical to the unmemoized version (figure 4). Only the function for the composite part changes. It now checks a hash table for a previously stored value, if it is there, then the cached value is used. Otherwise, the cost and mass is computed and then stored in the table for reference in the future. The hash table, keyed on part names, is provided by the module `TextToRefanyTable` which is part of the standard distribution of Modula-3.

The `REFANY` type (a pointer to anything) does subvert the type system. Run-time type checks are performed by the `NARROW` operation to insure that the reference is to an object of the appropriate type. It is possible to implement a generic hash type suitable for values of any type and then instantiate it for the type `Rcrd` instead of using `REFANY`. The compiler will then prevent any value other than `Rcrd` from being inserted in hash table. But it also requires an interface file be created that contains the type `Rcrd`.

Obtaining polymorphism this way is more burdensome than, say, in the programming language ML. The syntactic overhead would be worth the trouble in Modula-3, if a generic memoization module could be implemented for any function. Indeed this is simple to accomplish in a type-safe manner (at least for integer or string keys), despite remarks to the contrary [1, page 183]. It is utterly simple in ML although it brushes with an unhappy combination of universal polymorphism and references. Weak types [11] as in Standard ML of New Jersey are a sufficient solution for most purposes.

```
MODULE Memo EXPORTS Main;  (* compute cost and mass of a composite part *)
IMPORT Text, Wr, Stdio, Fmt;          (* Standard library modules *)
IMPORT TextToRefanyTable, RefanyList; (* Additional, standard modules *)
IMPORT Schema;                        (* Module for access to parts database *)

(*  Maintain a hash table of cost, mass pairs.  The keys are component
    names (type Text) which are assumed to be unique.                  *)
TYPE
    Rcrd = RECORD cost: Schema.Dollars; mass: Schema.Grams END;
VAR
    table := TextToRefanyTable.New (initialSize := 100);

PROCEDURE CostMassBase = (* same as before *)
PROCEDURE CostMass = (* same as before *)

PROCEDURE CostMassComp (
    READONLY comp: Schema.CompositePart;
    VAR cost: Schema.Dollars;
    VAR mass: Schema.Grams;
) =
    VAR
        value: REFANY;
        subtotal_cost: Schema.Dollars;
        subtotal_mass: Schema.Grams;
        list: RefanyList.T;
    BEGIN
        IF table.in (comp.name, value) THEN
            cost := NARROW(value,REF Rcrd).cost;
            mass := NARROW(value,REF Rcrd).mass;
        ELSE
            cost := comp.assembly_cost;
            mass := comp.mass_increment;
            list := comp.uses;
            WHILE list # NIL DO
                WITH
                    use = NARROW (list.first, Schema.Use)
                DO
                    CostMass (use.component, subtotal_cost, subtotal_mass);
                    INC (cost, use.quantity * subtotal_cost);
                    INC (mass, use.quantity * subtotal_mass);
                END;
                list := list.tail
            END;
            EVAL table.put (comp.name, NEW (REF Rcrd, cost:=cost, mass:=mass));
        END;
    END CostMassComp;

BEGIN
     (* same main program as before *)
END Memo.
```

Figure 7: Memoized version of computing cost and mass