The Modula-3 Type System

Luca Cardelli[*]
Jim Donahue[†]
Mick Jordan[†]
Bill Kalsow[*]
Greg Nelson[*]

Abstract

This paper presents an overview of the programming language Modula-3, and a more detailed description of its type system.

1 Introduction

The design of the programming language Modula-3 was a joint effort by the Digital Systems Research Center and the Olivetti Research Center, undertaken with the guidance and inspiration of Niklaus Wirth. The language is defined by the *Modula-3 Report* [3], and is currently being implemented by the Olivetti Research Center. This paper gives an overview of the language, focusing primarily upon its type system.

Modula-3 is a direct descendent of Mesa [8], Modula-2 [14], Cedar [5], and Modula-2+ [9, 10]. It also resembles its cousins Object Pascal [13], Oberon [15], and Euclid [6]. Since these languages already have more raw material than fits comfortably into a readable fifty-page language definition, which we were determined to produce, we didn't need to be inventive. On the contrary, we had to leave many good ideas out.

Instead of exploring new features, we studied the features from the Modula family of languages that have proven themselves in practice and tried to simplify them and fit them into a harmonious language. We found that most of the successful features were aimed at one of two main goals: greater robustness, and a cleaner, more systematic type system.

[*]DEC Systems Research Center (SRC)
[†]Olivetti Research Center

This paper begins with an overview of the language and then focuses on three aspects of its type system: the uniform description of type compatibility in terms of a subtype relation, the use of structural equivalence, and an extension of the type system to support object-oriented programming.

2 Language Overview

One of our main goals was increased robustness through safety from unchecked runtime errors — forbidden operations that violate an invariant of the runtime system and lead to an unpredictable computation.

A classic unchecked runtime error is to free a record that is referenced by active pointers. To avoid this danger, Modula-3 follows Cedar, Modula-2+, and Oberon by automatically freeing unreachable records. This affects the type system, since the type rules for references must be strict enough to make garbage collection possible at runtime.

Another well-known unchecked runtime error is to assign to the tag of a variant record in a way that subverts the type system. Distinguishing subversive assignments from benign assignments in the language definition is error-prone and arbitrary. The objects and classes first introduced by Simula [2] and adopted in Oberon and Object Pascal are more general than variant records, and they are safe, so we have discarded variant records and adopted objects.

In addition to being safer than variant records, objects types allow a measure of polymorphism for data structures like lists, queues, and trees. For example, a procedure that reverses a list object can safely be applied both to lists of integers and to lists of reals. All Modula-3 objects are references (unlike in C++ [12]). Modula-3 allows only single inheritance (unlike Owl [11]).

Generally the lowest levels of a system cannot be programmed with complete safety. Neither the compiler nor the runtime system can check the validity of a bus address for a peripheral device, nor can they limit the ensuing havoc if it is invalid. This presents the language designer with a dilemma. If he holds out for safety, then low level code will have to be programmed in another language. But if he adopts unsafe features, then his safety guarantee becomes void everywhere. In this area we have followed the lead of Cedar and Modula-2+ by adopting a small number of unsafe

features that are allowed only in modules that are explicitly labeled unsafe. In a safe module, the compiler guarantees the absence of unchecked runtime errors; in an unsafe module, it is the programmer's responsibility to avoid them.

From Modula-2+ we adopted exceptions. An exception exits all procedure call levels between the point at which it is "raised" and the point at which it is "handled". Exceptions are a good way to handle any runtime error that is not necessarily fatal. The alternative is to use error return codes, but this has the drawback that programmers don't consistently test for them. In the Unix/C world, the frequency with which programs omit tests for error returns has become something of a standing joke. Instead of breaking at an error, too many programs continue on their unpredictable way. Raising an exception is a more robust way to signal an error, since it will stop the computation unless there is an explicit handler for it.

Naturally we retained modules, which are separate program units with explicit interfaces. But we relaxed the Modula-2 rule that there be a one-to-one correspondence between interfaces and the modules that implement them. A module can implement a collection of interfaces; an interface can be implemented by a collection of modules.

We also retained opaque types, which hide the representation of a type from its clients. In Modula-3, as in some but not all implementations of Modula-2, variables with opaque types must be references. If the hidden representation changes but the interface remains the same, client modules will not need to be reprogrammed, or even recompiled. A type that is not opaque is called concrete. It is possible to reveal some but not all of the structure of a type, by declaring it to be an "opaque subtype" of a given concrete object type.

The concurrency features of Modula-2 provide runtime support for coroutines. In Modula-3 we have upgraded these features to support threads of control that can be executed concurrently on a multiprocessor. The features are a simplified version of the Mesa extensions to Hoare's monitors [4, 7] whose formal semantics have been specified in Larch [1]. Waiting, signaling, and locking a monitor have Hoare's semantics, but the requirement that a monitored data structure be an entire module is relaxed: it can be an individual record or any set of variables instead. The programmer is responsible for acquiring the appropriate lock before accessing the monitored data.

The language provides a syntactic construct for acquiring a lock, executing a statement, and releasing the lock. Except for this statement, the concurrency features are all specified by means of a "required interface", which is just like an ordinary interface except that all Modula-3 implementations must implement it. Thus concurrency adds little linguistic complexity.

Modula-3 provides a few convenience features that are not provided by Modula-2: default values for procedure arguments, keyword parameters in procedure calls, constructors for record and array values, and the ability to specify an initial value for a variable at the point of its declaration.

3 The Subtype Relation

Modula-3 is "strongly-typed". Ideally, this means that the value space is partitioned into types, variables are restricted to hold values of a single type, and operations are restricted to apply to operands of a fixed sequence of types. In actuality, things are more complicated. A variable of type **[0..9]** can safely be assigned to a variable of type **INTEGER,** but not vice versa. Operations like absolute value apply both to **REAL**s and **INTEGER**s**,** instead of to a single type (overloading). The types of literals (for example, **NIL**) may be ambiguous. The type of an expression may be determined by how it is used (target-typing). Type mismatches may cause automatic conversions instead of errors (as when a fractional real is rounded upon assignment to an integer).

We adopted several principles in order to keep Modula-3's type system as uniform as possible. First, there are no ambiguous types or target-typing: the type of every expression is determined only by its subexpressions, not by its use. Second, there are no automatic conversions. In some cases the *representation* of a value changes when it is assigned (for example, when assigning to a field of a packed record) but the abstract value itself is transferred without change. Third, the rules for type compatibility are defined in terms of a single subtype relation, written "**<:**". A naive plan for doing this goes as follows:

- define **T <: U** by rules relating to the syntax of **T** and **U**;
- define, for each type, the set of values of that type.

in such a way that these definitions satisfy

- Value set rule: **T <: U** if and only if every value of type **T** is a value of type **U**;
- Natural assignment rule: A **T** is assignable to a **U** if and only if **T <: U**.

This plan would lead to a highly uniform type system, but unfortunately it is too simple. For example, the natural assignment rule would forbid the assignment of an **INTEGER** to a **[0..9]**; but the conventional policy is to allow such an assignment, compiling a runtime check. We have no doubts that the conventional policy is the best one, so the natural assignment rule will not do. Any assignment satisfying the natural assignment rule is allowed, but in addition there are more liberal rules for ordinal types (integers, enumerations, and subranges), references, and arrays. These will be described below.

We were also forced to drop half of the value set rule: if **T <: U**, then every value of type **T** is also a value of type **U,** but the converse does not hold. This still provides a criterion for checking that a syntactic subtype rule is consistent with the semantics of the types involved, but it allows us to leave

out some subtype relations that are logically possible but pragmatically unattractive, because they would force the implementation to do too much work.

We will now describe the subtype rules for each class of types.

3.1 Ordinal types.

Subrange types are subtypes of their "base" types, since each member of a subrange is also a member of the corresponding base type:

[n..m] <: INTEGER if **n** and **m** are integers
[a..b] <: E if **a** and **b** are from the enumeration type **E**

Moreover, two subrange types are in subtype relation when their respective sets of values are in inclusion relation:

[a..b] <: [c..d] if **[a..b]** is a (possibly empty) subset of **[c..d]**

Note that partially overlapping subranges are completely unrelated.

3.2 Set types.

For the subtype rule for sets we simply use the value set rule:

SET OF T <: SET OF T' if **T <: T'**

This rule is very natural, although open to the objection that it requires the implementation to convert between representations for some assignment operations.

3.3 Reference types.

A reference type is either *traced* or *untraced.* A member of a traced reference type is traced by the garbage collector; that is, the implementation stores its referent in a system-managed storage pool, determines at runtime when all traced references to it are gone, and then reclaims its storage. A member of an untraced reference type is not traced by the garbage collector.

The type **REF T** is the type of all traced references to variables of type **T**; the type **UNTRACED REF T** is the type of all untraced references to variables of type **T**. The type **REFANY** is the type of all traced references; the type **ADDRESS** is the type of all untraced references; and the type **NULL** is the type containing only **NIL**.

The subtype rules for reference types are again determined by the value set rule:

```
NULL <: REF T <: REFANY
NULL <: UNTRACED REF T <: ADDRESS
```

Notice that the value **NIL** is a member of all reference types. This does not mean that the type of **NIL** is ambiguous: its type is **NULL**, which is assignable to all reference types by the natural assignment rule.

The **TYPECASE** statement can be used to determine the referent type of a variable of type **REFANY**, but there is no corresponding operation for variables of type **ADDRESS**. This difference reflects the fact that traced references must be tagged for the benefit of the garbage collector.

Untraced references are provided for several reasons. Low-level code may require pointers to device control blocks that do not reside in the system storage pool; linking with code that was compiled from another language may require pointers that are not valid traced references; and untraced references can provide significant performance advantages. Most operations on untraced references are type-safe. However, reclaiming the storage for an untraced reference is a potential unchecked runtime error, and so is not allowed in safe modules.

Object types are also reference types, but their subtyping rules will be described in a later section.

3.4 Procedure types.

Modula-3's procedure types are very similar to those in Modula-2, consisting essentially of a signature specifying the result type and the mode and type of each parameter. There are some minor differences: a Modula-3 procedure signature also specifies the set of exceptions that can be raised by the procedure, and allows the formal parameters to be named and given default values.

The subtype rule for procedure types T and U is:

T <: U if:

- **T** and **U** have the same number of parameters, and corresponding parameters have the same type and mode.
- **T** and **U** have identical return types, or neither has a return type.
- The exception set of **U** contains the exception set of **T**.

The reader may wonder why we did not follow the well-known "arrow rule", in which (writing **T -> U** for the type of all functions from type **T** to type **U**):

(T -> U) <: (T' -> U')
if **T' <: T** and **U <: U'**

The arrow rule cannot be used for **VAR** parameters, since they are in a sense both arguments and results. Even for value parameters and results the rule has undesirable consequences. Suppose for example that **T <: U** and that **T** is a procedure that takes a **U**, while **q** is a procedure variable declared to take

a **T**. The arrow rule allows the assignment **q := P**, since **P** is less "choosy" than **q**. It follows that the actual of type **U** that the compiler produces in the calling sequence to q must also be a valid actual of type **T**, since this will be expected by the body of **P**.

Thus if the arrow rule is used for procedure types, then whatever representation is used for variables of type **U** must also be used for variables of any subtype of **U**. This policy would rule out biased implementations of subrange types, for example. It is incompatible with the subtype rule given previously for sets. It would mean that fixed arrays passed by value would have to be treated like open arrays, that is, with an additional integer specifying the length. None of these consequences is decisively bad, but the arrow rule is not decisively good. We decided not to break with convention.

On the other hand, the subtype rule for procedures does not require the exception sets to be equal. This generality has no undesirable consequences for the implementation.

For convenience in handling procedure variables, **NIL** is also allowed as a procedure; thus we have the additional rule:

```
NULL <: PROCEDURE(A): B RAISES E
```

for any arguments **A**, result type **B**, and exception set **E**.

3.5 Packed types.

TYPE T = BITS n FOR U declares that type **T** has the same values as **U**, but record fields and array elements of type **T** will occupy exactly **n** bits. The subtyping rules for packed types are:

```
BITS n FOR T <: T
T <: BITS n FOR T
```

These rules are natural consequences of that fact that **T** and **U** have the same values. They make it possible to assign unpacked values to packed fields, and vice versa. It may seem surprising that **T** and **U** can be subtypes of one another without being identical, but this is appropriate when distinct types represent the same set of values.

3.6 Array types.

As in Modula-2, array types can be *fixed* or *open.* The length of a variable with a fixed array type is determined at compile time. The length of a variable with an open array type is determined at run time, when the variable is allocated or bound. It cannot be changed thereafter. Assignments are allowed between fixed and open arrays, with a run-time check that the lengths agree.

TYPE T = ARRAY I OF E declares **T** to be the type of fixed arrays with index type **I** and element type **E**. The index type must be an ordinal type. The subtype rule is:

```
ARRAY I OF T <: ARRAY J OF T
   if NUMBER(I) = NUMBER(J)
```

i.e. the arrays must have identical sizes and element types. Notice that the rule requires that the element types be identical, even though the value set rule would only require that the element type on the left be a subtype of the element type on the right. For example, consider:

```
TYPE
   T = ARRAY [0..999] OF [0..255];
   U = ARRAY [0..999] OF INTEGER;
```

Every sequence of a thousand integers in the range **[0..255]** is a sequence of a thousand integers, so the value set rule would require **T <: U**. But this would require complicated conversions to implement assignment and parameter passing, at least if **T** is represented differently from **U**, as is likely in many implementations. This complexity is the main reason that we dropped half of the value set rule.

Another point to note about the array subtype rule is that the domain types **I** and **J** don't need to be the same; they only need to have the same length. An array *value is* a sequence; an array *variable* consists of a value together with a method of indexing it: indexes are automatically decreased by the lower bound of the index set of the variable's type. Consequently the set of values of an array variable depends only on the length of the index set, and the subtyping rule above is consistent with the half of the value set rule that we are keeping. The advantage of this approach is that it allows all open arrays to have lower bound zero, which reduces bookkeeping at runtime. This may seem overly parsimonious, but the approach comes from Modula-2, where it has worked well.

The declaration **TYPE T = ARRAY OF E** declares **T** to be an open array type. The values of **T** are sequences of variables of type **E**. Open array variables are always indexed by integers starting at zero.

Obviously we need the rule

```
ARRAY I OF T <: ARRAY OF T
```

which allows a fixed array actual to be bound to an open array formal. Since Modula-3 allows multidimensional open arrays, we also need the rules

```
ARRAY J OF ARRAY I OF T
   <: ARRAY OF ARRAY OF T

ARRAY OF ARRAY I OF T
   <: ARRAY OF ARRAY OF T
```

These don't follow from the first rule, because in general the array rule requires that the elements types be identical. Generalizing to *n* dimensions, we obtain the following rule, which subsumes the previous three:

ARRAY I_1 OF ... ARRAY I_n OF T
<: (ARRAY OF)^n T

where the **I_i** are ordinal types or omitted. (Omitted **I**'s create open array types.)

Finally, the relation **<:** can be defined as the smallest reflexive and transitive relation that satisfies the rules presented above (and the rules for objects in Section 5).

3.7 Assignment rules.

A type **T** *is assignable* to a type **U** if one of the following conditions apply.

- **T <: U** (The natural assignment rule).
- **T** and **U** are ordinal types with at least one member in common.
- **U <: T** and **T** is an array type or reference type (including an object type, but excluding **ADDRESS** in safe modules).

In the first case, no run-time error is possible, since if **T** is a subtype of **U**, then every **T** is a **U**.

In the second case, a conventional range check is made to ensure that the particular **T** is a member of **U**.

The third case allows, for example, assigning a **REFANY** to a **REF T**. It also allows assigning an **ARRAY OF T** to an **ARRAY I OF T**. In this case a run-time check is required either on the type of the reference or on the length of the array.

The third rule is unconventional: in Cedar, Modula-2+, and Oberon, the rules for references allow a supertype to be assigned to a subtype only by using an explicit **NARROW** operation. But this strictness with references is somewhat inconsistent with the lenient rule for ordinal types. Furthermore, based on our survey of Modula-2+ programs, the conventional rule does not seem to make programs safer or more readable.

4 Type identity

Two types are identical if their definitions are the same when expanded; that is, when all names in the type definition are replaced by their definitions. In the case of recursive types, the expansion is infinite. In other words, Modula-3 uses structural equivalence, while Modula-2 uses name equivalence. (The term "name equivalence" is a misnomer: it doesn't mean that types are the same only if they have the same name; it means that each occurrence of a type constructor produces a new type. But it's a popular misnomer, so we'll use it.)

This decision may be surprising. Of the languages mentioned in the introduction, only Euclid uses structural equivalence. It seems at first that structural equivalence is worse for the programmer, since it weakens typechecking by introducing the danger of accidental type coincidences, and worse for the implementation, since it requires a non-trivial computation to determine whether two types are structurally equivalent. So why not stick with name equivalence?

The objection that structural equivalence weakens typechecking by creating accidental type coincidences has some truth in it, but the truth is more complicated than it first appears. For example, consider

```
TYPE
  Subrange1 = [0..255];
  Subrange2 = [0..255];
  Ref1 = POINTER TO INTEGER;
  Ref2 = POINTER TO INTEGER;
```

In Modula-2, these declarations produce four distinct types. But although all types are created distinct, some types are more distinct than others. A variable of type **Subrange1** can be assigned to a variable of type **Subrange2**, since the assignment rule for ordinal types is based on the structure of the type. A variable of type **Ref1** cannot be assigned to a variable of type **Ref2**, since the assignment rule for references requires type identity, and ignores the structure of the type.

We have met name-equivalence purists who get uneasy about this, and even try to change the rules to prevent assignments between **Subrange1** and **Subrange2**. After all, it certainly is true that assignments between **Subrange1** and **Subrange2** are sometimes bugs, and to let them slip by the compiler seems like a concession of defeat by all who believe in static typing. But this leads to type systems in which a **[0..10]** can't be assigned to a **[0..11]**, or to an **INTEGER.** This is very awkward, and probably impractical.

There is a fundamental trade-off between convenience and safety. If you want the convenience that a **[0..255]** automatically inherits all the attributes of an **INTEGER**, then you face the danger that you may accidentally use an **INTEGER** attribute that is not an attribute of the type represented by this instance of **[0..255]**. Modula-2 already has a mechanism for hiding attributes of a type, namely the opaque type machinery. It seems like a mistake for a subrange declaration to be doing the work of an opaque type declaration. So name-equivalence purists can relax: if a programmer erroneously assigns a Subrange1 to a **Subrange2** and complains that the type system let it through, they can tell her that she should have used an opaque type.

If this argument applies to **Subrange1** and **Subrange2**, why not to **Ref1** and **Ref2**? In Modula-3, the rule for assigning references is based on the subtype relation (like all assignment rules). Because of objects, Modula-3 reference types have a rich subtype structure, just like the ordinal types. The subtype rules make a **Ref1** a subtype of **Ref2**, and

therefore assignable to **Ref2**, whether they are distinct types or not.

Of course, a language with structure-based assignment rules can still use name equivalence. For example, in Modula-2 the types **Subrange1** and **Subrange2** are distinct, even though they are assignable. The results are a little odd: consider passing an actual parameter of type **Subrange1** to a formal of type **Subrange2**. In Modula-2, this is legal for a value parameter, but not for a variable parameter. This seems more of a quirk than a useful protection.

In other words, the more structure-based assignment rules, the weaker the argument that name equivalence prevents accidental type coincidences. Since Modula-3's type system is based on a subtype relation, this argument for retaining name equivalence was not persuasive.

In contrast, there is a strong argument for switching to structural equivalence, which is that structural equivalence makes sense between types that occur in different programs, while name equivalence makes sense only between types that occur in the same program. This advantage becomes significant when type safety is extended to distributed systems (via remote procedure call) or to permanent data storage systems.

For example, DEC SRC's Topaz system includes a package called **Pickles** for storing typed data on the disk. The call **Pickle.Write(r, f)** writes the data structure referenced by **r** into the file **f**, preserving any circularities, substructure sharing, and the types of the records involved. The preserved data is called a "pickle". The call **Pickle. Read(r, f)** sets the reference **r** to a reconstruction of the value pickled in the file **f**. (The run-time information that makes this possible consists of a a single "typecode" that identifies the type of each reference, which needs to be maintained for the garbage collector, anyway.) The question now arises: when is it type-safe for a pickle written by program **A** from a variable of type **T** to be read by another program **B** into a variable of type **U**?

With structural equivalence, the answer is obvious: the operation is type-safe if **T** and **U** are the same type. that is, if they have the same structure. Without structural equivalence, there is no satisfactory answer. Requiring that **T** and **U** have the same name doesn't work, since in different programs different types can have the same name. Requiring that **T** and **U** have the same structure (that is, using name equivalence within a program but structural equivalence for pickles) doesn't work, since it would allow two structurally equivalent references with distinct typecodes to be pickled and then read back into two references with identical typecodes. Requiring that **T** and **U** have the same name and the same structure works after a fashion; it is the current policy for the pickles package. But it has serious drawbacks. It means that changing the name of a type can invalidate a previously-created pickle. Since names have to be generated arbitrarily for anonymous types that appear in pickles, a pickle can also be invalidated just by reordering type declarations or by giving a name to a previously anonymous type. This is not just a theoretical problem, as programmers who have been bitten by it can testify.

5 Object types

The object types of Modula-3 are essentially Simula classes. The challenge we faced is to integrate them into the type system so that they fit well with the existing procedure and reference types. This section first motivates the two essential aspects of object types, inheritance and methods, then describes how they fit together in Modula-3, and finally sketches an efficient implementation.

5.1 Inheritance.

Consider the type declarations

```
TYPE
  A = REF RECORD a: REAL END;
  AB = REF RECORD a: REAL; b: BOOLEAN END;
```

Loosely speaking, a value **x** has type **AB** if it is the address of a word in memory containing an **a** field of type **REAL** followed by a word containing a **b** field of type **BOOLEAN**. Similarly, a value **x** has type **A** if it is the address of a word in memory containing an **a** field of type **REAL**. Thus every **AB** is an **A**; that is, loosely speaking, **AB <: A**.

In fact, in any conventional implementation, passing an **AB** actual to a procedure whose formal is of type **A** is safe and meaningful: the procedure operates on the **a** field of the record, without disturbing the **b** field.

This example illustrates the basic idea of inheritance of object types. In Modula-3, the type constructor **OBJECT** is like **REF RECORD**, but while the referent of a **REF RECORD** must consist exactly of the fields declared in the record type, the referent of the object type may have additional fields not mentioned in the object type. Also, the **OBJECT** type constructor can be used to extend an object type with additional fields as well as to create a new type from scratch. For example, to achieve the subtype relation **AB <: A**, the types above should be declared:

```
TYPE A = OBJECT a: REAL END;
TYPE AB = A OBJECT b: BOOLEAN END;
```

Here is an example of inheritance used to produce a reusable queue implementation. First, the interface:

```
TYPE
  Queue = RECORD head, tail: QueueElem END;
  QueueElem = OBJECT link: QueueElem END;

PROCEDURE
  Insert(VAR q: Queue; x: QueueElem);
  Delete(VAR q: Queue): QueueElem;
  Clear(VAR q: Queue);
```

The implementation of the procedures relies only on the **link** field of a **QueueElem**; it does not depend on any additional fields that might be present in particular subtypes. The implementation is obvious and will not be listed. Here is an example client:

```
TYPE IntQueueElem =
  QueueElem OBJECT val: INTEGER END;

VAR
  q: Queue;
  x: IntQueueElem;

...
Clear(q);
NEW(x, val := 6);
Insert(q, x);
...
x := Delete(q)
```

Passing **x** to **Insert** is safe, since every **IntQueueElem** is a **QueueElem**. Assigning the result of **Delete** to **x** cannot be guaranteed valid at compile-time, but the assignment will produce a checked runtime error if the source value is not a member of the target type. Thus **IntQueueElem** bears the same relation to **QueueElem** as **[0..9]** bears to **INTEGER**. Notice that the runtime check on the result of **Delete(q)** is not redundant, since other subtypes of **QueueElem** can be inserted into **q**.

5.2 Methods.

We begin with a simple example: how to implement a closure, which is simply a procedure bundled up with an argument record. For example, the formal parameter to **Thread.Fork**, which creates a new thread of control, is a closure. The first definition that comes to mind is:

```
TYPE Closure =
  OBJECT proc: PROCEDURE(self: Closure) END;
```

where the idea is that each subtype of **Closure** will extend the record with whatever additional data fields are appropriate to that subtype. In this representation, a closure **cl** is activated by calling **cl.proc(cl)**.

To test this definition, let us try to build a closure which, when activated, will compute and print the greatest common divisor of 111 and 259:

```
TYPE GCDClosure =
  Closure OBJECT n, m: INTEGER END;

PROCEDURE PrintGCD(cl: GCDClosure);
  BEGIN Print(GCD(cl.n, cl.m)) END PrintGCD;

VAR gcd: GCDClosure;

NEW(gcd,
  proc := PrintGCD, n := Ill, m := 259)
```

Unfortunately, the initialization of **gcd.proc** in the last line is illegal, since the declared argument type for **gcd.proc** is an arbitrary closure, while **PrintGCD** demands a **GCDClosure**. Even if Modula-3 had used the arrow rule for procedure subtyping, storing a choosy procedure value into a permissive procedure variable would not be type-safe.

It is awkward to work around this: we have to change **PrintGCD** to take an arbitrary closure and narrow it at runtime to a **GCDClosure**. (By "narrowing", we mean checked runtime type conversion.) The code would have to look like this:

```
PROCEDURE PrintGCD(cl: Closure);
  VAR gcl: GCDClosure := cl;
  BEGIN
    Print (GCD(gcl.n, gcl.m))
  END PrintGCD;
```

It is irritating as well as awkward, since in the program at hand it actually is safe to store the choosy procedure value in the permissive variable. The reason is that the only argument that the program ever supplies to **cl.proc** is **cl** itself. Given this, it is easy to see that the initialization of **gcd.proc** to **PrintGCD** "ought to be" type-safe: at the time it is compiled, the type of **gcd** is known to be not just a **Closure** but also a **GCDClosure**. By assumption, **gcd** is the only value that will ever be passed to **gcd.proc**, so it is all right for it to demand a **GCDClosure**.

This problem illustrates the typechecking aspects of the role of "methods" in object-oriented programming. The idea behind methods is that a general operation **P** is applied to a specific object **v** by calling a version of **P** that is customized for **v** (called **v**'s **P** method). That is, **P(v, ...)** simply translates to **v.P(v, ...)**. The **Closure** example is the special case in which there is only one method and the method's only argument is the object itself.

This method approach is most useful in conjunction with inheritance, since the suite of methods for a particular subtype of **v** will generally require extra data fields in addition to the data fields of the supertype. But, as we can see from the **Closure** example, the subtype methods will always need to be declared to accept objects of the supertype, and to narrow their arguments to the subtype at runtime. Thus errors that could be caught at compile time will not be caught until run time, which is unfortunate.

5.3 Objects in Modula-3.

The solution to the problem is to extend the type system to support object-oriented programming. Here's how this is done in Modula-3.

An object is either **NIL** or a reference to a data record paired with a method suite, which is a record of procedures that will each accept the object as a first argument. We will just describe traced objects; untraced objects, which are produced by adding the keyword **UNTRACED** to the type declaration, are entirely analogous.

The signatures of the initial methods of the method suite are determined by the object type, but the method suite can contain additional methods, just like the data record. Methods are simply procedures; there is no separate space of method values.

Since the only way to call a method in an object's method suite is to pass the object itself as the first argument, the first parameter can be of any type that contains the object. The rest of the method's signature must determine a subtype of the method declaration in the object type. More precisely, a procedure **p** satisfies a method declaration with signature **sig** for an object **x** if **p** is **NIL** or if:

- **p** is a top-level procedure whose first parameter has mode **VALUE** and whose type contains the value **x**, and
- if **p**'s first parameter is dropped, the resulting procedure type is a subtype of the procedure type determined by **sig**.

Notice that this definition allows the type of the first parameter of the method to vary with the type of the object containing the method. The notion of "satisfies" is used to define the set of values of an object type. First we consider the declaration of an object type without a supertype, which has the form:

```
TYPE T =
  OBJECT
    FieldList
  METHODS
    MethodList
  END
```

where **FieldList** is a list of field declarations, exactly as in a record type, and **MethodList** is a list of method declarations. Each method declaration has the form:

```
m sig := proc
```

where **m** is an identifier (the method name), **sig** is a procedure signature, and **proc** is a top-level procedure constant.

An object **x** is a member of the type **T** if it is **NIL** or a traced reference to a data record that contains the fields declared in **FieldList**, possibly followed by other fields, paired with a method suite that contains procedures that satisfy the method declarations in **MethodList**, possibly followed by other procedures.

The "**:= proc**" is optional. If present, it specifies a default method value used when allocating objects of type **T**; if absent, the default method value is **NIL**.

Using methods, the type **Closure** would be defined like this:

```
TYPE Closure = OBJECT METHODS proc() END;
```

The rest of the example needs no change. The initialization of the **proc** method to **PrintGCD** is allowed, since the type of the first parameter to the method (**GCDClosure**) is a supertype of the type of the object being allocated (also **GCDClosure**).

A consequence of this design is that the method signatures are statically determined by an object's type (except for the first argument), but the method values are not determined until the object is allocated. The values cannot be changed thereafter.

The declaration of an object type with a supertype has the form:

```
TYPE T =
  Supertype OBJECT
    FieldExtension
  METHODS
    MethodRevision
  END
```

where **Supertype** is an object type, **FieldExtension** is a list of additional field declarations, and **MethodRevision** is a list of additional declarations and method overrides of the form:

```
m : = proc
```

where **m** is the name of a method of the supertype and **proc** is a top-level procedure that is a legal default for method **m** in type **T**. Each method override specifies that **proc** is the default value used for method **m** when allocating objects of type **T**. If a method is not overridden, its default in **T** is the same as its default in the supertype.

An object is a member of the type **T** if its data record contains the fields of the supertype, followed by the fields declared in **FieldExtension**, possibly followed by other fields; and its method suite contains procedures that satisfy the method declarations in the supertype, followed by procedures that satisfy the method declarations in **MethodRevision**, possibly followed by other procedures.

Notice that all Modula-3 methods are "virtual methods". If it is known that a certain method will have a constant value for all objects of some type, then it might as well be declared as an ordinary procedure in the interface containing the type declaration.

The subtype rule for objects is simply the value set rule:

```
NULL <: T OBJECT ... END <: T
```

```
OBJECT ... END <: REFANY
UNTRACED OBJECT ... END <: ADDRESS
```

That is, **NIL** is a member of every object type, and every supertype contains its subtypes. Also, every object is a reference, traced or untraced.

5.4 Notation and examples.

If **r** is an object, then **r.f** designates the data field named **f** in **r**'s data record. If **m** is one of **r**'s methods, then the expression **r.m(...)** denotes the procedure application (**r**'s **m** method)**(r,...)**. If **T** is an object type and **m** is the name of one of **T**'s methods, then **T.m** denotes **T**'s default **m** method. The last notation is convenient when a subtype method must invoke a default method of one of its supertypes.

As an example, consider the following declarations:

```
TYPE
  A = OBJECT a: INTEGER; METHODS p() END;
  AB = A OBJECT b: INTEGER END;

PROCEDURE Pa(self: A) = ... ;
PROCEDURE Pab(self: AB) = ... ;

VAR a: A; ab: AB; ...
```

Obviously **AB <: A**. Since neither **A** nor **AB** has default method values, the method values must be specified when the objects are allocated. The procedures **Pa** and **Pab** are suitable values for the **p** methods of objects of types **A** and **AB**. For example:

```
NEW(ab, p := Pab)
```

creates an object with an **AB** data record and a method that expects an **AB**; it is an example of an object of type **AB**. Similarly,

```
NEW(a, p := Pa)
```

creates an object with an **A** data record and a method that expects an **A**; it is an example of an object of type **A**. A more interesting example is:

```
NEW(ab, p := Pa)
```

which creates an object with an **AB** data record and a method that expects an **A**. Since every **AB** is an **A**, the method is not too choosy for the object in which it is placed. The result is a valid object of type **AB**. In contrast,

```
NEW(a, p := Pab)
```

attempts to create an object with an **A** data record and a method that expects an **AB**; since not every **A** is an **AB**, the method is too choosy for the object in which it is placed. The result would not be a member of the type **AB**, so this call to **NEW** is a static error.

Here is an example that illustrates the use of default method values and method overrides:

```
TYPE Window =
  OBJECT
    extent: Rectangle
  METHODS
    mouse(e: ClickEvent) := IgnoreClick;
    expose(e: ExposeEvent) := IgnoreExpose
  END;

TYPE TextWindow =
  Window OBJECT
    text: Text.T;
    style: TextWindowStyle
  METHODS
    expose := ExposeTextWindow
  END;
```

If no methods are specified when an object of type **TextWindow** is allocated, its mouse method will be **IgnoreClick** and its expose method will be **ExposeTextWindow**. These procedures must be de-clared elsewhere. The procedure **ExposeTextWindow** can demand a **TextWindow** as its first parameter, but **IgnoreExpose** and **IgnoreClick** must accept any **Window**.

5.5 Implementation.

To solidify the preceding ideas we sketch one possible implementation of objects.

An object can be represented as the address of the first word of its data record. The preceeding word stores an object header containing a type code unique to the object type. These type codes are small integers; there is one of them for each object type and for each traced reference type. (A similar object header can be used for all records in the heap, whether they are objects or not.) The word before the object header stores a reference to the method suite for the object. An advantage of this scheme is that if the object has no methods, this word can be omitted. It also allows objects to share method suites, which will be the common case.

If **o** is an object, **d** one of its data fields, and **m** one of its methods, then in this representation:

```
o.d          is   Mem[o + d]
o.m          is   Mem[Mem[o – 2] + m]
TYPECODE(o)  is   Mem[o – 1]
```

where we assume that fields and methods are represented by offsets in the natural way.

The more interesting problem is to efficiently test if an object **o** has type **T**, as is required for narrowing and typecase statements.

The simplest implementation of narrowing main-tains an array **st** indexed by typecode; **st[tc]** is the typecode for the supertype of the object type whose typecode is **tc**, or **NIL** if there is no supertype. To test if **o** is a **T**, a loop is used to compute whether the typecode for **T** appears in the sequence

```
TYPECODE(o), st[TYPECODE(o)],
  st[st[TYPECODE(o)]], ... NIL
```

Let us call this sequence the *supertype path* for **o**'s type, and its length the *depth* of **o**'s type. Faster implementations of narrowing exploit the observation that the depth of each type is determined at compile time, and can therefore be stored with the corresponding typecode. Thus if the typecode for **T** appears in the supertype path for a type **U**, it does so at the position **depth(U) - depth(T)**. This means that narrowing can be implemented in constant time, if the supertype path for each type is represented as a sequential array. Since supertype paths are not usually too long, this is an attractive strategy. In the unusual case of an object type with a very long supertype chain, only a prefix of the chain, up to some maximum length, would be stored sequentially. If at runtime the difference in depth exceeds the length of the sequentially stored prefix of the chain, the implementation must fall back on the linked list.

References

[1] A.D. Birrell, J.V. Guttag, J.J. Horning, R. Levin. Synchronization Primitives for a Multiprocessor: A Formal Specification. *Operating Systems Review* 21 5, November 1987. Also published as SRC Research Report 20, August 1987.

[2] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin.* Auerbach, Philadelphia PA, 1973.

[3] Luca Cardelli, Jim Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Report.* Digital Systems Research Center, SRC-31, 1988.

[4] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM* 17 10, October 1974.

[5] Butler W. Lampson. A Description of the Cedar Language. Xerox Palo Alto Research Center, CU-83-15, December 1983.

[6] Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the Programming Language Euclid. Xerox Palo Alto Research Center, CSL81-12, October 1981.

[7] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM* 23 2, February 1980.

[8] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Xerox Palo Alto Research Center, CSL-78-1, February 1978.

[9] Paul Rovner, Roy Levin, and John Wick. On Extending Modula-2 For Building Large, Integrated Systems. Digital Systems Research Center, SRC-3, January 1985.

[10] Paul Rovner. Extending Modula-2 to Build Large, Integrated Systems. *IEEE Software* 3 6, November 1986.

[11] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. Trellis Object-Based Environment Language Reference Manual. DEC Eastern Research Lab, DEC-TR-372, 1985.

[12] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[13] Larry Tesler, Apple Computers. Object Pascal Report. Structured Language World 9 3, 1985.

[14] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, Third Edition, 1985.

[15] N. Wirth. From Modula to Oberon and The Programming Language Oberon. Institut fur Informatik, ETH Zurich 82, September 1987.