Twelve Changes to Modula-3
19 Dec 90

The Modula-3 committee has made twelve final changes to the language definition, described in this message.

On behalf of the Modula-3 committee I would like to thank Bob Ayers, Michel Gangnet, David Goldberg, Sam Harbison, Christian Jacobi, Nick Maclaren, Eric Muller, and Thomas Roemke for their helpful comments on these changes. ---Greg Nelson

Contents

Section 1. List of changes

1.1 The language will be extended to support generic interfaces and modules. The detailed semantics of generics are in Section 3, below.

1.2 In addition to REAL and LONGREAL the language will support the new floating point type EXTENDED. New required interfaces will allow clients to use IEEE floating point if the implementation supports it. The behavior of the interfaces is also defined for non-IEEE implementations. Listing of these interfaces, and other details, are in Section 4, below.

1.3 The default raises clause will be the empty set instead of the set of all exceptions. RAISES ANY will be used to indicate that a procedure can raise any exception.

1.4 The sentence:

> The declaration of an object type has the form
>
> TYPE T = ST OBJECT FieldList METHODS MethodList END
>
> where ST is an optional supertype, FieldList is a list of field declarations, exactly as in a record type, and MethodList is a list of "method declarations" and "method overrides".

will be changed to

> The declaration of an object type has the form
>
> TYPE T = ST OBJECT Fields METHODS Methods OVERRIDES Overrides END
>
> where ST is an optional supertype, Fields is a list of field declarations, exactly as in a record type, Methods is a list of "method declarations" and Overrides is a list of "method overrides".

The syntax for individual method declarations and individual method overrides remains the same.

1.5 The semantics of method overrides supplied at NEW time will be defined by the following rewriting:

NEW(T, m := P)

is sugar for

NEW(T OBJECT OVERRIDES m := P END).

As a consequence, the method overrides are restricted to procedure constants, and the methods of an object are determined by its allocated type. It is no longer necessary to refer to "T's default m method", you can just say "T's m method".

1.6 On page 48, the last sentence in the section "Constant Expressions" will be changed from

"All literals are legal in constant expressions; procedure constants are not"

to

"Literals and top-level procedure constants are legal in constant expressions"

Procedure application remains illegal in constant expressions, except for the required procedures in the Word interface.

1.7 The word "not" will be removed from the sentence "T.m is not a procedure constant" on page 39, and the grammar will be changed to allow the syntax T.m as a method default.

1.8 The prohibition against NEWing an opaque object type will be removed. The procedures Thread.NewMutex and Thread.NewCondition will be removed from the Thread interface and replaced by comments to the effect that a newly-allocated Mutex is in the unlocked state and a newly-allocated Condition has no threads waiting on it.

1.9 The following sentence will be added to the "revelations" section:

> In any scope, the revealed supertypes of an opaque type must be totally ordered by the subtype relation. For example, in a scope where it is revealed that T <: S1 and that T <: S2, it must also be revealed either that S1 <: S2 or that S2 <: S1.

1.10 The sentence

> The pragma <*EXTERNAL L*> precedes a procedure declaration to indicate that the procedure is implemented in the language L, or precedes an interface to indi cate that the entire interface is implemented in the language L.

will be changed to

> The pragma <* EXTERNAL N:L *> precedes an interface or a declaration in an interface to indicate that the entity it precedes is implemented by the language L, where it has the name N. If ":L" is omitted, then the implementation's default ex ternal language is assumed. If "N" is omitted, then the external name is determined from the Modula-3 name in some implementation-dependent way.

1.11 The result type of the built-in function TYPECODE will be changed from INTEGER to CARDINAL.

1.12 Changes to the syntax of text, character, and integer literals, as follows: On page 59, "Token Productions", the lines

> CharLiteral = "'" (PrintingChar | Escape) "'"
>
> TextLiteral = DQUOTE { PrintingChar | Escape } DQUOTE

will be changed to

> CharLiteral = "'" (PrintingChar | Escape | DQUOTE) "'"
>
> TextLiteral = DQUOTE { PrintingChar | Escape | "'" } DQUOTE

The effect is to allow "Don't" instead of "Don\'t" and '"' instead of '\"'. In the section on integer literals, we will add the words

> If no base is present, the value of the literal must be at most LAST(INTEGER); if an explicit base is present, the value of the literal must be less than 2^Word.Size, and its interpretation as a signed integer is implementation-dependent. For example, on a sixteen-bit two's complement machine, 16_ffff and -1 represent the same value.

Section 2. Rationale

2.1 In regard to generics: several programmers have invented ad-hoc schemes for working around the absence of generics. We have found a simple design that seems to be in the Modula spirit, is easy to implement, and has essentially no impact on the rest of the language.

2.2 In regard to floating-point revisions: Jorge Stolfi and Stephen Harrison have reported on their use of Modula-3 for floating-point graphics computations, and David Goldberg has given us a critique of Modula-3 from the point of view of a numerical analyst. We have used this feedback to try to make Modula-3 better for floating-point computations, an issue that was not taken too seriously in the original design.

2.3 In regard to the default raises clause, RAISES { } is far more frequent than RAISES ANY, so it should be the default. This change is not backward-compatible: conversion will require

> Finding occurrences of RAISES { } and deleting them. This is easy to do, and is not essential, since RAISES { } is still meaningful and legal, although now redundant.
>
> Finding procedures without raises clauses and adding RAISES ANY (or, more likely, noticing that they were incorrectly annotated and that the correct clause is RAISES { } in which case they can be left alone). Since RAISES ANY is rare, this change should be necessary in very few places, e.g., for mapping functions.

2.4 In regard to the explicit OVERRIDES keyword: several programmers have accidentally declared a new method when they meant to override an existing one, strongly suggesting that the current syntax does not distinguish between methods and overrides strongly enough. This change is not backwards compatible, but existing programs can be converted easily, by

sorting the method declarations in front of the method overrides and adding the word "OVERRIDES" between the two groups.

2.5 In regard to defining the semantics of method overriding at NEW time by syntactic sugar: the advantage of this change is that the method suite becomes a constant function of the type. That is, we no longer have to talk about "T's default m method", we can just say "T's m method". For example, this is convenient for the implementation of type-safe persistent storage (pickles), which can recreate an object's method suite trivially, by asking the runtime system for the method suite associated with the type. (This is in fact what is done by both the Olivetti and the SRC pickles packages: adopting the rewriting semantics above will make both packages correct.)

The obvious cost of the change is that a procedure variable can no longer be used as a method override. However, no programs are known to use this facility. Furthermore, a strong argument can be made that this facility is not useful. For example, consider the following program, which does use a procedure variable as a method override:

```
PROCEDURE New(
  h: PROCEDURE(t: Table.T, k: Key.T): INTEGER)
: Table.T =
BEGIN
    ...
  NEW(Table.T, hashMethod := h)
END New.
```

This is poor code: it would be better to make the hash method a procedure-valued field in the table, since the usual argument for making something a method rather than a procedure data field doesn't apply here: New cannot be used to construct a subtype of Table.T with a hash method specific to that subtype, because New's signature requires that h accept any Table.T as an argument. Consequently there is no flexibility gained by using a method. Furthermore, if the hash procedure is stored in the data part of the object, hash tables with different hash procedures can share method suites, and thus save space.

This change is not strictly backwards-compatible, but as no programs are known to override methods at NEW time with non-constant procedures, it is expected to be backwards compatible in practice.

2.6 In regard to the change that allows procedure constants in constant expressions: this allows procedure constants as default parameters, which is a convenience. And it does not seem to be difficult for implementations, or any less principled to allow procedure constants in constant expressions than to allow TEXT literals.

2.7 In regard to allowing "T.m" as a procedure constant: you could already write T.m to denote the m method of type T, but because this expression was not considered a procedure constant, it couldn't be used as a method default. That usually meant that an interface exporting T would also have to export each of T's methods by name, since somebody defining a related class might reuse some of the methods. Thus this change allows less cluttered interfaces.

Allowing T.m as a procedure constant raises the following question:

Are U and V the same after

```
TYPE T = OBJECT METHODS m() := P END
TYPE
  U = T OBJECT METHODS m := P END;
  V = T OBJECT METHODS m := T.m END;
```

Answering "yes" would be hard on the implementation, since although in this case the identity of T.m with P is manifest, this would not be true in general.

So we will define U and V to be different, on the grounds that the expanded form of U contains "P" where the expanded form of V contains "(expanded form of T).m". With respect to structural equivalence, the implementation should treat the occurrence of "T.m" the same as it would treat a record field "m: T".

2.8 In regard to NEW(T) for opaque types T: If an opaque type requires initializations over and above what can be done with default field values, then the implementor of the type must provide a procedure for doing the initialization. He might provide a New procedure that allocates, initializes, and returns a value of the type, but this New procedure couldn't be used by anybody implementing a subtypes of the opaque type, since it allocates the wrong type of object. Thus when you declare an opaque type that is intended to be subclassed, you owe it to your clients to provide an init routine that takes an object that has already been allocated, or to comment that no initialization in necessary beyond what is automatically provided by default field values. In either case, the effect of NEWing the opaque type will be well-specified in the interface, and the old rule against it is revealed as an attempt to legislate style.

In fact, the old rule interfered with a style that seems attractive: for each object type T, define a method T.init that initializes the object and returns it. Then the call

```
NEW(T).init(args)
```

allocates, initializes, and returns an object of type T. If T is an subtype of some type S, and the implementer of S uses the same style, then within T.init(self, ...) there will be a call of the form EVAL S.init(self, ...) to initialize the supertype part of the object. Note that this style involves NEWing the opaque object type T.

2.9 In regard to the requirement that the revealed supertypes of an opaque type must be totally ordered by the subtype relation: this rule was present in the original version of the report, and somehow got deleted from the revised version, probably by an editing error, without the committee ever deliberately rescinding it. The advantage of the rule is that the information about an opaque type in a scope is determined by a single type, its "<:-least upper bound", rather than by a set of types. This simplifies the compiler; in fact, both the SRC and Olivetti compilers depend on this rule. Theoretically this is not a backwards-compatible change, but since it is bringing the language into conformance with the two compilers that are in use, obviously it won't require conversion by clients.

2.10 In regard to the EXTERNAL pragma: in producing the Modula-3 interfaces to the X library the need for renaming as part of the external declaration was acute.

2.11 Changing the result type of TYPECODE to CARDINAL is no burden on the implementation, and the guarantee that negative integers cannot be confused with typecodes is often convenient.

2.12 Allowing "Don't" instead of "Don\'t" makes programs more readable. Allowing integer literals whose high-order bit is set is useful both in graphics applications, where masks are being constructed, and in low-level systems code, where integers are being loopholed into addresses that may be in the high half of memory.

Section 3. Details of generics

Before describing the generics proposal proper, we describe two minor changes that are associated with generics: allowing renamed imports, and allowing text constants to be brands.

3.1 Allowing renamed imports.

IMPORT I AS N means "import the interface I and give it the local name N". The name I is not introduced into the importing scope.

The imported interfaces are all looked up before any of the local names are bound; thus

```
IMPORT I AS J, J AS I;
```

imports the two interfaces I and J, perversely giving them the local names J and I respectively.

It is illegal to use the same local name twice:

```
IMPORT J AS I, K AS I;
```

is a static error, and would be even if J and K were the same.

The old form IMPORT I is short for IMPORT I AS I.

FROM I IMPORT X introduces X as the local name for the entity named X in the interface I. A local binding for I takes precedence over a global binding for I. For example,

```
IMPORT I AS J, J AS I; FROM I IMPORT X;
```

simultaneously introduces local names I, J, and X for the entities whose global names are J, I, and J.X, respectively.

3.2 Allowing text constants as brands.

The words from the section on Reference types:

> ... can optionally be preceded by "BRANDED b", where b is a text literal called the "brand".

will be changed to

> ... can optionally be preceded by "BRANDED b", where b is a text constant called the "brand".

This allows generic modules to use an imported text constant as the brand.

3.3 Generics proper.

In a generic interface or module, some of the imported or exported interface names are treated as formal parameters, to be bound to actual interfaces when the generic is instantiated.

A generic interface has the form

GENERIC INTERFACE G(F_1, ..., F_n); Body END G.

where G is an identifier that names the generic, F_1, ..., F_n is a list of identifiers, called the formal imports of G, and Body is a sequence of imports followed by a sequence of declarations, exactly as in a non-generic interface. An instance of G has the form

INTERFACE I = G(A_1, ..., A_n) END I.

where I is the name of the instance and A_1, ..., A_n is a list of "actual" interfaces to which the formal imports of G are bound. The semantics are defined precisely by the following rewriting:

INTERFACE I; IMPORT A_1 AS F_1, ..., A_n AS F_n; Body END I.

A generic module has the form

GENERIC MODULE G(F_1, ..., F_n); Body END G.

where G is an identifier that names the generic, F_1, ..., F_n is a list of identifiers, called the formal imports of G, and Body is a sequence of imports followed by a block, exactly as in a non-generic module. An instance of a G has the form

MODULE I EXPORTS E = G(A_1, ..., A_n) END I.

where I is the name of the instance, E is a list of interfaces exported by I, and A_1, ..., A_n is a list of actual interfaces to which the formal imports of G are bound. "EXPORTS E" can be omitted, in which case it defaults to "EXPORTS I". The semantics are defined precisely by the following rewriting:

MODULE I EXPORTS E; IMPORT A_1 AS F_1, ..., A_n AS F_n; Body END I.

Notice that the generic module itself has no exports or UNSAFE indication; they can be supplied only when it is instantiated.

For example, consider

```
GENERIC INTERFACE Dynarray(Elem);
  (* Extendible arrays of Elem.T, which can be any type  except an open array type. *)
    TYPE T = REF ARRAY OF Elem.T;
    PROCEDURE Extend(VAR v: T);
  (* Extend v's length, preserving its current contents. *)
END Dynarray.
```

```
GENERIC MODULE Dynarray(Elem);

PROCEDURE Extend(VAR v: T) =
VAR w: T;
BEGIN
  IF v = NIL OR NUMBER(v^) = 0 THEN
        w := NEW(T, 5)
        ELSE
        w := NEW(T, NUMBER(v^) * 2);
        FOR i := 0 TO LAST(v^) DO w[i] := v[i] END
        END;
        v := w
        END Extend;

END Dynarray.
```

To instantiate these generics to produce dynamic arrays of integers:

```
INTERFACE Integer; TYPE T = INTEGER END Integer.

INTERFACE IntArray = Dynarray(Integer) END IntArray.

MODULE IntArray = Dynarray(Integer) END IntArray.
```

Implementations are not expected to share code between different instances of a generic module, since this will not be possible in general.

There is no typechecking associated with generics: implementations are expected to expand the instantiation and typecheck the result. For example, if one made the following mistake:

```
INTERFACE String; TYPE T = ARRAY OF CHAR END String.

INTERFACE StringArray = Dynarray(String) END StringArray.

MODULE StringArray = Dynarray(String) END StringArray.
```

Everything would go well until the last line, when the compiler would attempt to compile a version of Dynarray in which the element type was an open array. It would then complain that the "NEW" call in Extend does not have enough parameters.

Section 4. Details of floating-point

The new built-in floating-point type EXTENDED will be added. The character "x" will be used in place of "d" or "e" to denote EXTENDED constants.

FIRST(T) and LAST(T) will be defined for the floating point types. In IEEE implementations, these are minus and plus infinity, respectively.

MOD will be extended to floating point types by the rule

x MOD y = x - y * FLOOR(x / y).

Implementations may compute this as a Modula-3 expression, or by a method that avoids overflow if x is much greater than y.

All the built-in operations that take REAL and LONGREAL will take EXTENDED arguments as well. (To make the language specification shorter and more readable, we will use the type class "Float" to represent any floating point type. For example:

+ (x,y: INTEGER) : INTEGER

(x,y: Float) : Float

The sum of x and y. If x and y are floats, they must have the same type, and the result is the same type as both.

LONGFLOAT will be removed and FLOAT will be changed to:

FLOAT(x: INTEGER; T: Type := REAL): T

FLOAT(x: Float; T: Type := REAL): T

Convert x to have type T, which must be a floating-point type.

Thus FLOAT(x) means the same thing it means today; FLOAT(x, LONGREAL) means what LONGFLOAT(x) means today.

We will add to the expressions chapter a note that the rounding behavior of floating-point operations is specified in the required interface FloatMode, as well as a note that implementations are only allowed to rearrange computations if the rearrangement has no effect on the semantics; e.g., (x+y)+z should not in general be changed to x+(y+z), since addition is not associative. The arithmetic operators are left-associative; thus x+y+z is short for (x+y)+z. The behavior of overflows and other exceptional numeric conditions will also be determined by the interface FloatMode. Finally, we will change the definition of the built-in equality operation on floating-point numbers so that it is implementation-dependent, adding a note that in IEEE implementations, +0 equals -0 and Nan does not equal Nan.

Modula-3 systems will be required to implement the following interfaces related to floating-point numbers. The interfaces give clients access to the power of IEEE floating point if the implementation has it, but can also be implemented with other floating point systems. For definitions of the terms used in the comments, see the IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std. 754-1985).

```
INTERFACE Real;
TYPE T = REAL;
CONST
   Base: INTEGER = ...;
  (* The radix of the floating-point representation for T *)
  Precision: INTEGER;
 (* The number of digits of precision in the given Base for T. *)
  MaxFinite: T = ...;
 (* The maximum finite value in T. For non-IEEE implementations,  this is the same as
            LAST(T). *)
  MinPos: T = ...;
 (* The minimum positive value in T. *)
 MinPosNormal: T = ...;
(* The minimum positive "normal" value in T; differs from MinPos  only for implementations
          with denormalized numbers. *)

END Real.

INTERFACE LongReal;
TYPE T = LONGREAL;
CONST
   Base: INTEGER = ...;
   Precision: INTEGER = ...;
   MaxFinite: T = ...;
    MinPos: T = ...;
    MinPosNormal: T = ...;
   (* Like the corresponding constants in Real. *)
END LongReal.

INTERFACE Extended;

TYPE T = EXTENDED;
```

```
CONST
   Base: INTEGER = ...;
   Precision: INTEGER = ...;
   MaxFinite: T = ...;
   MinPos: T = ...;
   MinPosNormal: T = ...;
   (* Like the corresponding constants in Real. *)
END Extended.

INTERFACE RealFloat = Float(Real) END RealFloat.
INTERFACE LongFloat = Float(LongReal) END LongFloat.
INTERFACE ExtendedFloat = Float(Extended) END ExtendedFloat.

GENERIC INTERFACE Float(Real);

TYPE T = Real.T;
(* The purpose of the interface is to provide access to the floating-point operations required or
   recommended by the IEEE floating-point standard. Consult the standard for the
   precise specifications of the procedures, including when their arguments are NaNs,
   infinities, and signed zeros, and including what exceptions they can raise. Our
   comments specify their effect when the arguments are ordinary numbers and no ex-
   ception is raised. Implementations on non-IEEE machines that have values similar
   to NaNs and infinities should explain how those values behave for IsNaN, Finite,
   etc. in an implementation guide *)
PROCEDURE Scalb(x: T; n: INTEGER): T;
(* Return x * (2 ** n). *)

PROCEDURE Logb(x: T): T;
(* Return the exponent of x. More precisely, return the unique n such that ABS(x) / Base ** n
   is in the range [1..Base-1], unless x is denormalized, in which case return MinExp-1, where
   MinExp is the minimum exponent value for T. *)
PROCEDURE ILogb(x: T): INTEGER;
(* Like Logb, but returns an integer, never raises an exception, and always returns the n such
   that ABS(x)/Base**N is in [1..Base-1] even for denormalized numbers. *)
```

```
PROCEDURE NextAfter(x, y: T): T;
(* Return the next representable neighbor of x in the direction towards y. If x = y, return x *)

PROCEDURE CopySign(x, y: T): T;
(* Return x with the sign of y. *)

PROCEDURE Finite(x: T): BOOLEAN;
(* Return TRUE if x is strictly between -infinity and +infinity. This always returns TRUE on
   non-IEEE machines. *)

PROCEDURE IsNaN(x: T): BOOLEAN;
(* Return FALSE if x represents a numerical (possibly infinite) value, and TRUE if x does not
   represent a numerical value. For example, on IEEE implementations, returns TRUE if x is a
   NaN, FALSE otherwise; on Vaxes, returns TRUE of x is a reserved operand, FALSE other
   wise. *)

PROCEDURE Sign(x: T): [0..1];
(* Return the sign bit x. For non-IEEE implementations, this is the same as ORD(x >= 0); for
   IEEE implementations, Sign(-0) = 1, Sign(+0) = 0. *)

PROCEDURE Differs(x, y: T): BOOLEAN;
(* RETURN (x < y OR y < x). Thus, for IEEE implementations, Differs(NaN, x) is always
   FALSE; for non-IEEE implementations, Differs(x,y) is the same as x # y. *) PROCEDURE
   Unordered(x, y: T): BOOLEAN; (* Return NOT (x <= y OR y <= x). For non-IEEE imple-
   mentations, this always returns FALSE. *)

PROCEDURE Sqrt(x: T): T;
(* Return the square root of T. Must be correctly rounded if FloatMode.IEEE is TRUE. *)

TYPE IEEEClass =
    {SignalingNaN, QuietNaN, Infinity, Normal, Denormal, Zero};

PROCEDURE Class(x: T): IEEEClass;
```

```
(* Return the IEEE number class containing x. On non-IEEE systems, the result will be Nor
   mal or Zero. *)
END Float.

INTERFACE FloatMode;
(* This interface allows you to test the behavior of rounding and of numerical exceptions. On
   some implementations it also allows you to change the behavior, on a per-thread basis. *)

CONST IEEE: BOOLEAN = ...;
(* TRUE for full IEEE implementations. *)

EXCEPTION Failure;

TYPE RoundingMode =
{MinusInfinity, PlusInfinity, Zero, Nearest, Vax, IBM370, Other};
(* Mode for rounding operations. The first four are the IEEE modes. *)

CONST RoundDefault: RoundingMode = ...;
(* Implementation-dependent: the default mode for rounding arithmetic operations, used by a
   newly forked thread. This also specifies the behavior of the ROUND operation in half-way
   cases. *)

PROCEDURE SetRounding(md: RoundingMode) RAISES {Failure};
(* Change the rounding mode for the calling thread to md, or raise the exception if this cannot
   be done. This affects the implicit rounding in floating-point operations. Generally this is pos
   sible only on IEEE implementations and only if md is an IEEE mode. *)

PROCEDURE GetRounding(): RoundingMode;
(* Return the rounding mode for the calling thread. *)
TYPE Flag =
  {Invalid, Inexact, Overflow, Underflow, DivByZero, IntOverflow, IntDivByZero};
  (* Associated with each thread is a set of boolean status flag recording whether the condition
     represented by the flag has occurred in the thread since the flag has last been reset. The
     meaning of the first five flags is defined precisely in the IEEE floating point standard;
     roughly they mean:
```

```
    Invalid = invalid argument to an operation.
    Inexact = an operation produced an inexact result.
    Overflow = a floating-point operation produced a result whose absolute value is too large to
      be represented.
    Underflow = a floating-point operation produced a result whose absolute value is too small to
      be represented.
    DivByZero = floating-point division by zero.
    The meaning of the last two flags is:
    IntOverflow = an integer operation produced a result whose absolute value is too large to be
      represented.
    IntDivByZero = integer DIV or MOD by zero. *)

CONST NoFlags = SET OF Flags {};

PROCEDURE GetFlags(): SET OF Flag;
(* Return the set of flags for the current thread *)

PROCEDURE SetFlags(s: SET OF Flag): SET OF Flag;
(* Set the flags for the current thread to s, and return their previous values. *)

PROCEDURE ClearFlag(f: Flag);
(* Turn off the flag f for the current thread. *)

EXCEPTION Trap(Flag);
TYPE Behavior = {Trap, SetFlag, Ignore};
(* The behavior of an operation that causes one of the flag conditions is either
    Ignore = return some result and do nothing.
    SetFlag = return some result and set the condition flag. For IEEE implementations,
      the result will be what the standard requires.
    Trap = possibly set the condition flag; in any case raise the Trap exception with the
      appropriate flag as the argument.
*)
```

```
PROCEDURE SetBehavior(f: Flag; b: Behavior) RAISES {Failure};
(* Set the behavior of the current thread for the flag f to be b, or  raise Failure if this cannot be done. *)

PROCEDURE GetBehavior(f: Flag): Behavior;
(* Return the behavior of the current thread for the flag f. *)

END FloatMode.
```