Extending Active Oberon with Enumeration Types

B. Kirk 11th April 2014

1 Introduction and Motivation

Active Oberon (AO) is a language which is part of the evolution of a family of languages: PASCAL, Modula-2, Oberon and Active Oberon: other languages such as Ada are also descended from PASCAL [1]. All are so called 'strongly typed' languages and place an emphasis on programming in a style that enables the compiler to make extensive checks to ensure that operations are applied to data in a consistent and appropriate way. What should be allowed is allowed and what shouldn't be allowed is detected at compile time and is forbidden.

Strongly typed languages tend to be used in application domains involving safety such as security, automation, transport and medical applications where testability during development and dependability during operation are of prime importance.

The evolutionary step from Modula-2 to Oberon [2] was motivated by the need to program 'object oriented' solutions to problems and also a quest for language simplification, achieved by removing current language features deemed unnecessary for the projects envisioned by Oberon's designers. This was implemented in Oberon and Active Oberon by the addition of extensible types [2, 4] and the removal of many features including enumeration types [3, 5].

The elimination of enumeration types rendered Oberon and Active Oberon incapable of providing compiler checks involving "scalar types [1]", i.e. groups of values, which are logically bound together, such as the state names/values of a state machine or colours of the rainbow etc. In this instance the compiler checkable program type safety was sacrificed in favour of simplicity of compiler implementation [3, 8]. See [10] for an alternative view providing a rationale for programming without enumerations in Oberon.

This document proposes an extension (addition) to the Active Oberon language to make representation of enumeration types possible in programs. For completeness it also proposes a means to make enumeration types extensible based on the scheme used in Modula-2 R10 [8].

2 Enumeration Types

The use of enumeration types provides type safety by ensuring that invalid values cannot be used for any variable or parameter of an enumeration type involving operations on variables of that type. In a language without enumeration types, or with "quasi" enumeration types (e.g. C) programmers must manually check that values are not out of range, but for large programs this becomes practically impossible, even for small programs it is difficult. For example the source of the Oberon System is littered with groups of CONST declarations which provide a type less and error prone substitute for enumeration types. For modern high integrity industrial and commercial programming this is certainly not good programming practice.

Two main objections have been levelled against enumeration types [10]:

1. Potential ambiguity of naming when importing an enumeration type from another module
2. Lack of type extensibility

An enumeration type is an ordinal type, it's valid values are defined by a list of identifiers. To provide a simple solution to the potential ambiguity problem all enumerated values are qualified with the type identifier. The identifiers are assigned ordinal values from left to right as they appear in the type definition. The ordinal value assigned to the leftmost identifier is always 0, the maximum number of identifiers in an enumeration is implementation dependent.

Suppose that this TYPE is declared in the exporting module

```
MODULE Graphics;
  TYPE Monochrome* = (black, white); (* ORD(black) has the value 0*)
```

After importing the type to this module the variable can be declared, and then used

```
MODULE Application;

  IMPORT Graphics;

  VAR pixel : Monochrome;

BEGIN
  pixel := Monochrome.white; (* qualified *)
```

When an enumeration type is imported from another module, then all the identifier names defined its original declaration are implicitly imported as well. The type's group of identifiers can be used unambiguously in the importing module due to their qualification when used, i.e. prefixing each imported identifier name with the name of the type that has been imported.

To provide extensibility of type declarations the scheme proposed in the specification of Modula-2 R10 [8] is used. An enumeration type may be defined as an extension of an existing type declaration by including within it the identifier of the base type as the first item in the enumerated value list suffixed by the "$" symbol. All enumerated values of the base type become legal values of the new type. But note that the base type is only *downwards compatible* with any extended types derived from it, extensions *are not upwards compatible* with their base type. This restriction exists because any value of the base type is always a legal value of any extension type derived from it, however not every value of an extension type is also a valid value of the base type.

For example:

```
TYPE Monochrome* = (black, white);
TYPE ColourRGB = (Monochrome$, red, blue, green)
TYPE ColourCYM = (Monochrome$, cyan, yellow, magenta)

VAR a:Monochrome; b,d:ColourRGB, c,e:ColourCYM;

BEGIN

    a:= white;
    b:= blue;
    c:= yellow;

    b:= a (*valid - value of b is now white *);
    c:= a (*valid - value of c is now white *);
    d:= b (*valid - value of d is now blue *);
    e:= c (*valid - value of e is now yellow *);

    a:= b (*invalid*)
    a:= c (*invalid*)
    b:= c (*invalid*)
    c:= b (*invalid*)
```

3 Built in language facilities

Some built language facilities are needed for clear and clean programming, typically

ORD	returns the integer value representing an identifier
PRED	returns the previous identifier in the enumerated sequence
SUCC	returns the succeeding identifier in the enumerated sequence
FIRST	returns the value of the first identifier in the enumerated sequence
LAST	returns the value of the last identifier in the enumerated sequence

4 Implementation Specifics

The following are implementation specific:

1. The value range supported for enumerations, i.e. max number of identifiers in an enumeration declaration. It is suggested that a ORD would return a 16 or 32 bit integer value depending on consistency with the current compiler implementation conventions. In practice using only an 8 bit integer may be too limiting.
2. Any jump tables, e.g. used for CASE statement lookups, could be sized base on the actual subrange of values in each specific declaration of an enumeration.
3. The behavior of PRED and SUCC for values going out of range needs to be defined e.g. wrap around OR runtime error trap OR limit, return the given value unchanged
4. The behavior for an attempted assignment out of range at runtime needs to be defined

This list is most probably incomplete, please add more items as they become apparent.

5 Active Oberon Syntax in EBNF with Enumeration Types

The syntax of Active Oberon given here is based on the documentation of the the Eigen Compiler Suite [6], with the permission of F Negele. The changes to the syntax to support enumeration types are added in italics.

This syntax needs to be reviewed by F Negele please!!

```
(* Active Oberon Syntax copied from [9] with Enumerated Types *)

Module = "MODULE" Identifier ["IN" Identifier] ";" [Imports]
                  {DeclSeq} Body Identifier ".".
Imports = "IMPORT" Import {"," Import} ";".
Import = Identifier [":=" Identifier] ["IN" Identifier].
         DeclSeq = "CONST" {ConstDecl ";"}
         | "TYPE" {TypeDecl ";"}
         | "VAR" {VarDecl ";"} | ProcDecl ";".
ConstDecl = Identifier ["*"] "=" ConstExpr.
TypeDecl = Identifier ["*"] "=" Type.
VarDecl = IdentList ":" Type.
ProcDecl = "PROCEDURE" ["{" "NORETURN" "}"] Signature ";"
           [{DeclSeq} Body Identifier].
Signature = ["&" | "~"] IdentDef [FormalPars].
FormalPars = "(" [FPSection {";" FPSection}] ")" [":" Type].
FPSection = ["VAR" | "CONST"] Identifier {"," Identifier} ":" Type.
Type = Qualident
  | "BOOLEAN" | "CHAR" | "SET" | "REAL" | "LONGREAL"
  | "SHORTINT" | "INTEGER" | "LONGINT" | "HUGEINT"
  | "ADDRESS" | "SIZE" | "WORD" | "LONGWORD"
  | "ARRAY" [ConstExpr {"," ConstExpr}] "OF" Type
  | "RECORD" [BaseType] [FieldList] "END"
  | "POINTER" ["{" "UNSAFE" "}"] "TO" Type
  | "OBJECT" [ [BaseType] {DeclSeq} Body]
  | "PROCEDURE" ["{" "NORETURN" "}"] [FormalPars]
  | "TYPE" "OF" Expr

(*Enumeration support - declare the type*)
  | “ENUM” EnumDecl.
EnumDecl = IdentDef “=” “(“ [QualIdent “$”] “,” IdentList “)”.

BaseType = "(" Qualident ")".
FieldDecl = [IdentList ":" Type].
FieldList = FieldDecl {";" FieldDecl}.
Body = StatBlock | "END".
StatBlock = "BEGIN" ["{"IdentList"}"] [StatSeq] "END".
StatSeq = Statement {";" Statement}.
Statement = [Designator ":=" Expr
  | Designator ["(" ExprList")"]
  | "IF" Expr "THEN" StatSeq {"ELSIF" Expr "THEN" StatSeq}
  ["ELSE" StatSeq] "END"
  | "CASE" Expr "DO" Case {"|" Case}
  ["ELSE" StatSeq] "END"
  | "WHILE" Expr "DO" StatSeq "END"
  | "REPEAT" StatSeq "UNTIL" Expr
  | "FOR" Identifier ":=" Expr "TO" Expr
  ["BY" ConstExpr] "DO" StatSeq "END"
```

```
  | "LOOP" StatSeq "END"
  | "WITH" Qualident ":" Qualident "DO" StatSeq "END"
  | "EXIT"
  | "RETURN" [Expr]
  | "WAIT" "(" Expr ")"
  | "AWAIT" "(" Expr ")"

(* Implementation defined behavior *)

  | "ASSERT" "(" Expr ["," ConstExpr] ")"
  | "HALT" "(" [ConstExpr] ")"
  | "NEW" "(" ExprList ")"
  | "DISPOSE" "(" Expr ")"
  | "INC" "(" Expr ["," Expr] ")"
  | "DEC" "(" Expr ["," Expr] ")"
  | "INCL" "(" Expr "," Expr ")"
  | "EXCL" "(" Expr "," Expr ")"
  | "COPY" "(" Expr "," Expr ")"
  | "GETPROCEDURE" "(" Expr "," Expr "," Expr ")"
  | "TRACE" "(" ExprList ")"
  | StatBlock].
Case = [CaseLabels { "," CaseLabels } ":" StatSeq].
CaseLabels = ConstExpr [".." ConstExpr].
ConstExpr = Expr.
Expr = SimpleExpr [Relation SimpleExpr].
SimpleExpr = ["+" | "-"] Term {AddOp Term}.
Term = Factor {MulOp Factor}.
Factor = Designator ["(" ExprList ")"] | "NIL"
  | Character | Number | String | Set
  | "(" Expr ")" | ["[" ExprList "]"] | "~" Factor
  | "CAP" "(" Expr ")" | "LOW" "(" Expr ")"
  | "ORD" "(" Expr ")" | "CHR" "(" Expr ")"
  | "ABS" "(" Expr ")" | "ENTIER" "(" Expr ")"
  | "LONG" "(" Expr ")" | "SHORT" "(" Expr ")"
  | "MIN" "(" Type ")" | "MAX" "(" Type ")"
  | "SIZE" "OF" Type | "ADDRESS" "OF" Designator
  | "ODD" "(" Expr ")" | "ASH" "(" Expr "," Expr ")"
  | "LEN" "(" Expr ["," Expr] ")"
  | "CAS" "(" Expr "," Expr "," Expr ")"

(* Enumeration support - ORD is already defined above*)
  | "SUCC" "(" Expr ")"  | "PRED" "(" Expr ")"
  | "FIRST" "(" Expr ")" | "LAST" "(" Expr ")".

Set = "{" [Element {"," Element}] "}".
Element = Expr [".." Expr].
Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | "IN" | "IS".
MulOp = "*" | "DIV" | "MOD" | "/" | "&".
AddOp = "+" | "-" | "OR".
Designator = Qualident { "." Identifier
  | "[" ExprList "]"
  | "^"
  | "(" Qualident ")" }.
ExprList = Expr {"," Expr}.
IdentList = IdentDef {"," IdentDef}.
Qualident = [Identifier "."] Identifier.
IdentDef = Identifier ["*" | "-"] ["{" "UNTRACED" "}"]
[":=" ConstExpr | "EXTERN" String].

(* end of syntax *)
```

6 References

[1] The Programming Language Pascal, 1972 , (see section 6.1.1, scalar types)
http://www.eah-jena.de/~kleine/history/languages/Wirth-PascalRevisedReport.pdf

[2] Type Extensions, *N. Wirth*, ACM Trans on Programming Languages and Systems, 10:2, 204-214, Apr. 1988
http://www.ethoberon.ethz.ch/books.html#Wir88c

[3] From Modula to Oberon, *N. Wirth*, Software - Practice and Experience, 18:7, 661-670, Jul. 1988
http://www.ethoberon.ethz.ch/books.html#Wir88a

[4] Extensibility in the Oberon System, Hans-Peter Mossenbock
https://www.cs.helsinki.fi/njc/njc1_papers/number1/inv_paper4.pdf

[5] The Oakwood Guidelines for Oberon-2 Compiler Developers, B Kirk (ed)
http://www.math.bas.bg/bantchev/place/oberon/oakwood-guidelines.pdf

[6] Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System, ETH Dissertation 22298 (PhD), Florian Negele, 2015

[7] Another view of Einstein's dictum:
"Everything should be made as simple as possible, but not any simpler."
http://knightsoftype.blogspot.co.uk/2013/12/einsteinian-simplicity-in-context-of.html

[8] Modula-2 Reloaded, see the Language Report at :
http://modula-2.info/m2r10/

[9] User Manual for Active Oberon, 9th December 2014, F Negele, Eigen Compiler Suite

[10] Programming Without Enumerations in Oberon
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.7983&rep=rep1&type=pdf

7 This Document's Change Log

If you make changes to this document please add a change record **at the top** of the list.

```
(*@ Change Log for: Extending Active Oberon with Enumeration Types

 #    Date (ymd)   By         Description

001   2016-04-11   BK         First draft 17:38
*)
```

<end of document>

160411 1145 : BK wish list for future
BYTE as in 'Oberon Station' Oberon
SHORTSET (8bit), SET916 bit?), LONGSET (32 bit?), HUGESET (64 bit set) or SET8, SET16, SET32, SET64
CASE used for discriminator of extended RECORD variants as in 'Oberon Station' Oberon