Adding an Authorization Dimension to Strong Type Checking

Scott N. Gerard
IBM Corporation
Rochester, MN

INTRODUCTION

Strongly typed languages, like Modula2, ensure that memory locations are interpreted in a consistent manner. I call this the structural dimension of the type system. I propose that an authorization (or capability, or rights) dimension be added to control who can use those memory locations. The structural dimension of a variable is the same for all procedures throughout a program, but different procedures may be given different authority for the same variable. Authorizations are similar to the `in`, `out` and `inout` parameter modes of ADA (tm), but are more comprehensive and apply to all identifiers.

Each identifier has a set of four authorizations:

```
TYPE
   AUTH = SET OF ( R, W, E, S );
```

R — (Read) Authority to read the current contents of an identifier. All identifiers used on the right-hand side of simple assignment statements must allow Read authorization.

W — (Write) Authority to overwrite the contents of an identifier. All identifiers used on the left-hand side of simple assignment statements must allow Write authorization.

E — (Execute) Authority to execute an identifier. All procedures normally allow Execute authorization, but there are interesting cases where this is not true.

S — (Structure) Authority to examine the structure of an identifier. The internal structure of a type is exported out of a module if, and only if, the type allows Structure authorization. That is, types with Structure authorization are transparent on export; types without Structure authorization are opaque on export. The only valid operations on identifiers without Structure authorization are `TSIZE`, `SIZE`, and `ADR`.

COMPILER CHANGES

There is no run-time overhead involved in implementing authorizations. All checks take place at compile time.

Each variable and type identifier declaration contains an optional `AUTH` set followed by the normal (structural) type information. Authorizations may decrease or stay the same, but they may never increase. Assume Type1 is declared with authorization Auth1, Type2 is declared with authorization Auth2 and Type1, and Type3 is declared with authorization Auth3 and Type2. Then the compiler must check that Auth2 is a proper or improper subset of Auth1, and Auth3 is a proper or improper subset of Auth2.

All built-in Modula2 types are implicitly defined with maximum authorization.

```
TYPE                                    (* compiler defined types *)
   INTEGER  = AUTH{R,W,S} WORD;
   CARDINAL = AUTH{R,W,S} WORD;
```

The compiler also must check that the authorization required by each formal parameter is a proper or improper subset of the authorizations available on the corresponding actual parameter. All Modula2 built-in operators become implicitly declared as pseudo-procedures. Special authorization checking code is not sprinkled throughout the compiler. For example, binary addition, pointer dereferencing and assignment are defined as:

```
PROCEDURE +  (     a,b: AUTH{R,S} NumericType): AUTH{R,S} NumericType;
PROCEDURE -> (     ptr: AUTH{R,S} PointerType): TargetType;
PROCEDURE := ( VAR lhs: AUTH{W} Type; rhs: AUTH{R} Type);
```

Function procedures have an authorization (usually AUTH{R,E,S}) which is different from the authorization of their returned value. The procedure's authorization controls whether it can be assigned to procedure variables, and whether it can be executed. Only after a procedure is determined to be executable does the returned value's authorization come into play.

There is one exception to the rule of never-increasing authority: the authorization for parameters of type MyMod.MyType are never checked on procedure calls which transfer control from outside module MyMod to a procedure inside MyMod. This allows MyMod to limit the authority given to client modules, yet still allows procedure MyMod.MyProc to manipulate them. However, the authorization in a procedure heading limits what can be done inside the procedure's body.

EXAMPLE

Let's examine a module for stacks of INTEGERs using authorizations.

```
DEFINITION MODULE StackADT;
EXPORT QUALIFIED Stack, Push, Pop, Empty, Init;
TYPE Stack: AUTH{} RECORD    (* a completely opaque type *)
            Top: [0..100];
            Elements: ARRAY [1..100] OF INTEGER;
            END;
PROCEDURE Push  (VAR S:    AUTH{R,W,S} Stack;
                     elem: AUTH{R,S}   INTEGER);
PROCEDURE Pop   (VAR S:    AUTH{R,W,S} Stack): AUTH{R,S} INTEGER;
PROCEDURE Empty (    S:    AUTH{R,S}   Stack): AUTH{R,S} BOOLEAN;
PROCEDURE Init  (VAR S:    AUTH{W,S}   Stack);
   END StackADT.


MODULE Program;
FROM StackADT IMPORT Stack, Push, Pop, Empty, Init;
VAR  S1,S2: Stack;
     i,j: INTEGER;
 BEGIN
   Init(S1);             (* don't check authorization on S1 here *)
   IF Empty(S1) THEN     (* don't check authorization on S1 here *)
      Push(S1,i)         (* don't check authorization on S1 here *)
   ELSE
      j := Pop(S1);      (* don't check authorization on S1 here *)
      END;
   S2 := S1;             (* illegal. can't read S1, can't write S2 *)
   S1.Top := 0;          (* illegal. Structure of S1 is not visible *)
   END Program.
```

Parameter elem of Push can be read as an INTEGER by the procedure body, but the body is prevented from writing it (or executing it as a procedure). Similarly, the returned value of Pop can

be read by the caller as an INTEGER, but cannot be written (or executed). Type `Stack` is declared with an empty authorization set. So, even though its structure is given in the definition module, all client modules are prevented from reading, writing, executing, or examining variables of type `Stack`. This is a strong form of opaque export. Variable `S1` does not have an AUTH set specified in its declaration, so it gets `AUTH{}` from `StackADT.Stack`. Variables `i` and `j` get `AUTH{R,W,S}` from INTEGER.

Each procedure should request the least amount of authority it needs to perform its function. This makes the procedure as widely usable as possible, explicitly declares the developers intentions, and ensures the procedure body does not violate those intentions.

The authorization of `S1` is not checked on any procedure call in `Program` because control is passing from outside module `StackADT` to a procedure inside `StackADT`. So its authorization may increase or decrease. Empty's procedure body can examine, but not change, parameter `S1`.

AUTHORIZATION INTERPRETATIONS

Many of the 16 authorization combinations are meaningful.

`AUTH{R,W,E,S}` normal procedure variables.

`AUTH{R, E,S}` normal procedure identifiers, and exported procedures variables which cannot be changed.

`AUTH{ W,E,S}` write-only procedure variables. Example: a procedure which can be changed, but no one can make a private copy.

`AUTH{ E,S}` procedure values which cannot be changed or copied.

`AUTH{R,W, S}` normal variables.

`AUTH{R, S}` read-only variables.

`AUTH{ W, S}` write-only variables. Examples: I/O output ports, imported random number seed variables, and variables which contain passwords (but see Enforcement below).

`AUTH{R,W }` opaque variable export in Modula2 today. This is similar to ADA's "private" export since assignment is allowed. (ADA also supports comparison on private types).

`AUTH{R }` opaque "constant".

`AUTH{ }` complete opaque export. Client modules are only authorized to "hold" these variables. All operations, including assignment, must go through procedures of the exporting module. This is the same as ADA's "limited private" types.

Note that Modula2 constants and read-only variables are not the same. Constants have a value which is known at compile time. Read-only variables may be evaluated at run time. The system clock is a good example of a read-only variable which is not a constant.

ENFORCEMENT

It is important to realize that authorizations only provide a reasonable level of protection. They can be easily subverted. The classic untagged variant record will do the trick.

```
TYPE
   Subversion = RECORD
      CASE BOOLEAN OF
         TRUE:  ( Opaque:     AUTH{}       INTEGER ) |
         FALSE: ( ReadWrite:  AUTH{R,W,S}  INTEGER )
         END
      END;
```

For the same reasons type (structure) transfer functions were added to Modula2, authorization transfer functions should be implemented for the rare occasions where it is justified.

In some implementations it may be possible to use the memory protection facilities of the underlying operating system to strictly enforce the read/write/execute authorizations. If so, the declared authorizations provide the necessary information.

USER DEFINED AUTHORIZATION SETS

There are many syntactic variations for specifying authorizations. One possibility is to simplify parameter declarations by making AUTH a real SET. Then developers can declare new authorization sets in addition to any predeclared ones. This would make programs much more readable.

```
TYPE                              (* compiler defined type *)
   AUTH = SET OF ( ReadAuth, WriteAuth, ExecAuth, StructAuth );

CONST
   In            = AUTH{ ReadAuth,                 StructAuth};
   Out           = AUTH{           WriteAuth,      StructAuth};
   InOut         = AUTH{ ReadAuth, WriteAuth,      StructAuth};
   ReadOnly      = AUTH{ ReadAuth,                 StructAuth};
   Opaque        = AUTH{ ReadAuth, WriteAuth                 };
   TotallyOpaque = AUTH{ };

VAR
   Stack:        Opaque RECORD ..... END;
   EndOfFile:    ReadOnly BOOLEAN;
```

OTHER IMPLICATIONS

Authorizations cover many other, more specialized, suggestions for improving Modula2.

There have been many proposals about parameter transmission. Authorizations are a more general concept than both pass-by-constant; and ADA's `in`, `out` and `inout` modes.

The existing pass-by-var and pass-by-value modes could be eliminated -- although this is not required. Technically, pass-by-VAR is equivalent to pass-by-reference. But the developer usually just means "I need to update the parameter." Pass-by-value/result would work as well; only the compiler writer should worry about such distinctions.

Pass-by-value is the same as passing the actual parameter to an anonymous variable with `ReadAuth`, and then using it to initialize a local variable. Pass-by-value is the one case of initialized variables in Modula2.

Pass-by-reference could be used throughout the compiler whenever the actual and formal are compatible, rather than just assignment compatible.

Authorizations directly answer the question of whether assignment and equality testing are defined on opaque types (a difference between Wirth2 vs. Wirth3, and ADA's private vs. limited private types). It supports both (`AUTH{R,W}` vs. `AUTH{}`). Authorizations allow the user to use opaque

export from local modules. This is not possible today because local modules do not come in definition/implementation pairs.

Modula2 does not support structured constants. This can be done with read-only variables. And module initialization sections can compute "constants" which are exported read-only variables.

Today, functions must never appear on the left-hand side of an assignment statement. This is to catch errors like

```
F(x) := 12;
```

But it also prevents expressions like

```
FPtr(x)-> := 12;
```

With the definition of the assignment and dereferencing pseudo- procedures given above, assignments to `F(x)` are still illegal, but assignments to `FPtr(x)->` are valid.

Authorizations allow the developer to declare opaque types in a definition module with a `TSIZE` different than `TSIZE(pointer)`. Although this is not recommended in general, it can be very useful in those rare situations where the type can never change.

LEFT-HANDED FUNCTION PROCEDURES

I have always been bothered by the asymmetry of the certain pairs of operations like: stack `Push/Pop`, queue `Enque/Deque`, file `Read/Write`, `Store/Retrieve` (e.g. routines to store and retrieve elements from a triangular matrix). `Pop` can be nicely written as a function with one stack parameter which returns one element. But `Push` must be written as a procedure with two parameters: the stack and the element to be pushed. But making `Push` a "left-handed" function solves the problem nicely.

```
element := Pop(Stack)
Push(Stack) := element;
```

A left-handed function is similar to a normal (right-handed) function but it appears on the left-hand side of assignment statements. The value from the right-hand side of the assignment is passed into the function body.

The difference between `Push` as left-handed function and as a no-handed procedure is purely syntactical; it provides no new capability. But then, neither do right-handed functions. I feel both dramatically increase the readability of the code because they show the programmer's intent so clearly.

`Push`, like `Pop`, now has an extra "anonymous" parameter. `Pop` RETURNs the value for this parameter; `Push` RETRIEVEs the value from this parameter. Authorizations naturally describe the difference. The anonymous parameter for right-handed functions has `ReadAuth` but not `WriteAuth`; for left-handed functions it has `WriteAuth` but not `ReadAuth`. An anonymous parameter with both `ReadAuth` and `WriteAuth` is invalid.

```
IMPLEMENTATION MODULE StackADT;

PROCEDURE Pop  ( VAR S: AUTH{R,W,S} Stack ): AUTH{R,S} INTEGER;
VAR element: INTEGER;
BEGIN
   element := Stack.Elements[Stack.Top];
   DEC( Stack.Top );
   RETURN element;
   END Pop;

PROCEDURE Push ( VAR S: AUTH{R,W,S} Stack ): AUTH{W,S} INTEGER;
VAR element: INTEGER;
BEGIN
   INC( Stack.Top );
   RETRIEVE element;              (* retrieve the anonymous parm *)
   Stack.Elements[Stack.Top] := element;
   END Pop;

END StackADT.
```

Matching actual parameters to formal parameters can also be looked at as a kind of assignment statement. It then makes sense to pass a "left-handed expression" to a formal which has `WriteAuth` but not `ReadAuth`.

```
PROCEDURE Proc  ( x: AUTH{R,   S} XType   ;
                  y: AUTH{R,W,S} YType   ;
                  z: AUTH{  W,S} ZType   );

PROCEDURE Right ( a,b,c: INTEGER ): AUTH{R,S} XType;
PROCEDURE Left  ( d,e,f: INTEGER ): AUTH{W,S} ZType;

Proc( Right(a,b,c), y, Left(d,e,f) );
```

is evaluated as:

```
xtemp := Right(a,b,c);
Proc( xtemp, y, ztemp );
Left(d,e,f) := ztemp;
```

Temporary variables are used for all formal parameters which have `ReadAuth` or `WriteAuth`, but not both (formals which have both `ReadAuth` and `WriteAuth` can only be satisfied by Modula2 variables). All `ReadAuth` parameters are evaluated to temporaries before the invocation, and all `WriteAuth` expression are evaluated afterwards.

The normal authorization checking between actual and formal parameters exactly describes which actual parameters can be left-handed functions, which can be right-handed functions, and which must be variables.

ACKNOWLEDGEMENTS

I would like to thank Paul Gunsch and Dick Orgass for their assistance on earlier drafts of this paper. I would also like to thank Gunsch for many interesting discussions about Modula2.

REFERENCES

1. N. Wirth, "Programming in Modula-2," 2nd edition, Springer-Verlag, 1983.
2. G. Fort and R. Wiener, "Modula-2: A Software Development Approach," John Wiley & Sons, Inc., 1985.
3. W. Waite and G. Goos, "Compiler Construction," Springer-Verlag, 1984.
4. B. Gaither, correspondence from the members, SIGPLAN Notices, March 1982.
5. A. Mayer, correspondence from the members, SIGPLAN Notices, March 1982.
6. M. Wilkes, "Empiric: A Sketch of a Programming Language Designed to Facilitate a Fine Grain of Protection," SIGPLAN Notices, August 1986.