EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

CERN - SL DIVISION

CERN-SL-2000-048 AP

Map creation and analysis via overloaded tools in FORTRAN 90

E. Forest*, F. Schmidt

In tracking codes there is the need to obtain, at run time, various machine quantities which depend parametically on things such as momentum or quadrupole strength. To this end we have overloaded (in FORTRAN 90) Berz' DA package [1] as well as the analysis library LieLib [2,3] which is based on this package and we have created polymorphic types. Runtime polymorphism is not interpretation as in COSY-INFINITY [4] and is more appropriate to large ring tracking codes. Consequently we have applied these tools to the code SixTrack [6].

paper presented at EPAC2000, 26-30 June 2000, Vienna, Austria

* KEK

Geneva, Switzerland

03.08.00

SL/Div Reps

Map Creation and Analysis via overloaded Tools in FORTRAN 90

E. Forest, KEK; F. Schmidt, CERN

Abstract

In tracking codes there is the need to obtain, at run time, various machine quantities which depend parametrically on things such as momentum or quadrupole strength. To this end we have overloaded (in FORTRAN90) Berz's DA-package [1] as well as the analysis library LieLib [2, 3] which is based on this package and we have created polymorphic types. Runtime polymorphism is not interpretation as in COSY-INFINITY [4] and is more appropriate to large ring tracking codes. Consequently we have applied these tools to the code SixTrack [6].

1 INTRODUCTION

A substantial amount of accelerator theory reduces to the calculation of a normal form. The normal form is a formal statement concerning the stability properties of the ring under consideration. For example, if one states that a linear system can be normalised into a rotation, and then this statement is mathematically equivalent to the so-called linear Courant-Snyder theory. Normal form concepts can be applied to the equations of motion or finite "s" maps. Because they are applicable to finite "s" maps the entire machinery of automatic differentiation, introduced by Berz to our field, can be used. Libraries such as our Lielib can be written to normalise these maps. Finally the normalising transformations can be propagated using the same tracking code that produced the maps in the first place.

There are two impediments to the usage of these techniques in accelerators. First there is a cultural barrier: unless one learns how to rephrase the standard theory in terms of finite maps, then the machinery of Berz is not usable. Map methods in rings, in conjunction with Lie methods, give us an elegant and powerful practical way to do linear lattice calculations which are easily extendible to nonlinear problems. Unless one is willing to sit down and (re)learn linear theory, then these finite "s" map methods will be difficult if not impossible to swallow.

Secondly, if one swims against the cultural current and learns the map methods, there is still a formidable task in dealing with FORTRAN77 libraries such as the "DA-package" and Lielib. Berz, first to notice this problem, started the COSY-INFINITY program in which he devised an interpreted language. The COSY language permits users to access Taylor series and manipulate them using a nicer syntax. For example, the addition of two Taylors A and B could become C=A+B rather than using a call to a subroutine such as CALL DAADD(A,B,C). COSY-INFINITY reads a file, which it then interprets. This solution is certainly acceptable to single pass systems but fits poorly into tracking codes for large rings.

Modern languages such as C++ and also FORTRAN90 permit operator overloading and thus allows one to use the syntax C=A+B in a compiled code. In addition, one would like to do "parameter dependence at run time." The authors had included this feature into their code SixTrack and DESPOT respectively in a very cumbersome way using the old FORTRAN77 "DA-package." Bengtsson, to our knowledge, was first to realize that in languages like C++, one could develop a delegated (run time) polymorphic type that permit precisely this kind of run time conversion. Remarkably, and somewhat contrary to public belief, such polymorphic types can be constructed in FORTRAN90 as well (see Szymanski [5]).

We decided to go back to Bengtsson's idea which consisted in linking directly Berz's package and Lielib to the appropriate polymorphic classes. This work could certainly have been done in C++ [8] ; we are not advocating any language over another.

2 THE POLYMORPHIC PACKAGE

The library is a hierarchical set of packages (modules in FORTRAN90) and they are displayed in Figure 1. The first modules definition.f90 contains definitions of types as the name indicates. For example, a type called taylor:

```
TYPE TAYLOR
INTEGER I             ! integer is a pointer in old da-package
TYPE (TAYLORLOW) J    ! taylorlow is the newda taylor series
END TYPE TAYLOR
```

Taylor is the type for a polynomial. The integer I points to the integer used by Berz in his package to locate a polynomial. The type Taylorlow illustrates the possibility of putting several TPSA libraries under the "hood."

The next 3 modules overload the old "TPSA-Package" of Berz and the physics library LIELIB. The basic TPSA operations are defined in TPSA.F90, basic maps and their concatenation are defined in TPSALIE.F90 and finally more advanced options such as normal forms and map factorisations are defined in TPSALIE_ANALYSIS.F90. Up to this point we have overloaded the basic functionality of the "DA-package" and LIELIB. A code using these tools only could resemble very much the COSY-INFINITY input file except that it requires compilation: it is not interpreted.

The first addition to this edifice was COMPLEX_TAYLOR.F90. Here we defined a complex polynomial as two polynomials of type taylor: the real and imaginary parts.

```
TYPE(COMPLEXTAYLOR)
TYPE (TAYLOR) R
TYPE (TAYLOR) I
END TYPE COMPLEXTAYLOR
```

The two final packages are REAL_POLYMORH.F90 and COMPLEX_POLYMORPH.F90. In these packages we define a real and a complex polymorphic type. We will

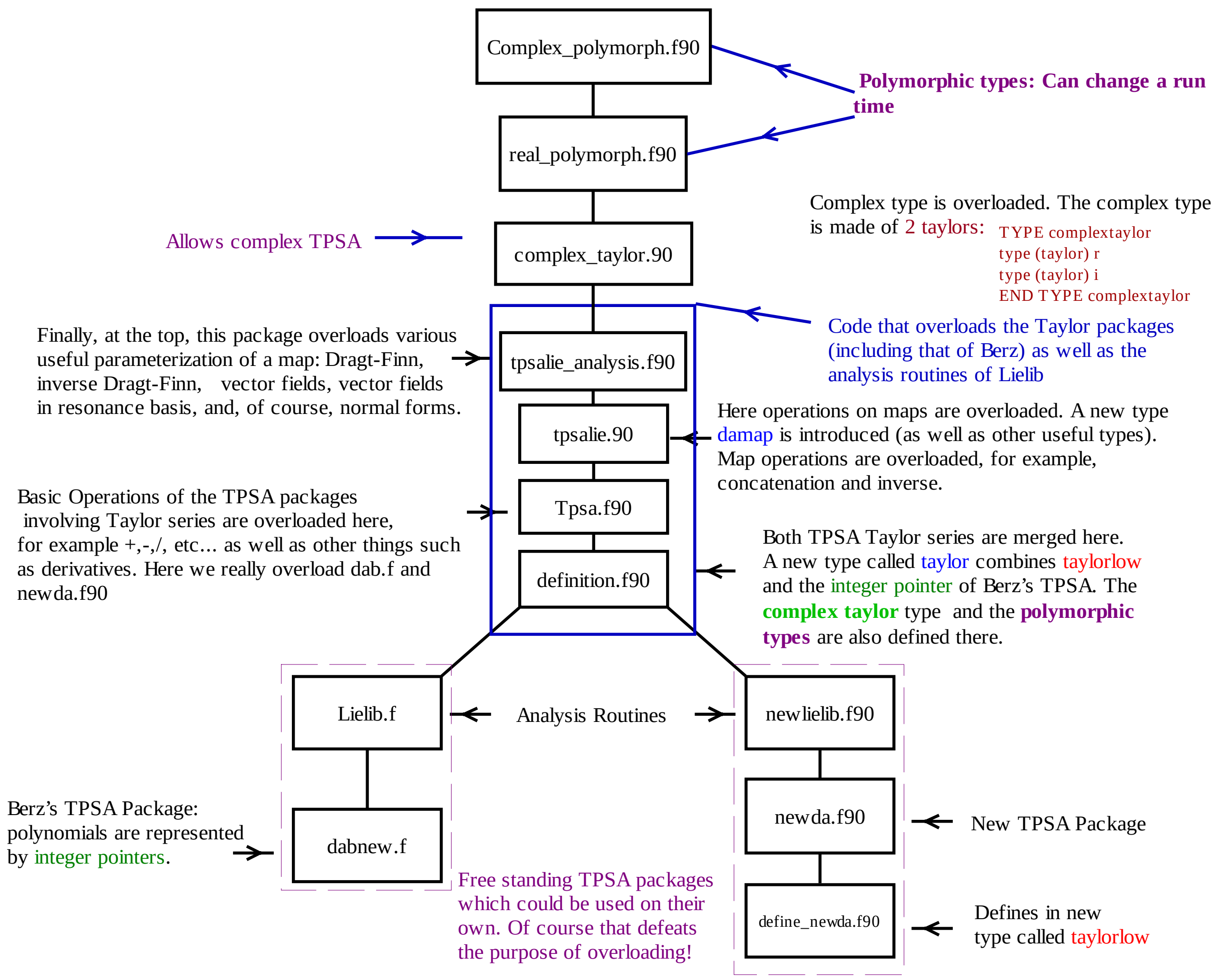
Complex_polymorph.f90
real_polymorph.f90
Polymorphic types: Can change a run time
Allows complex TPSA
complex_taylor.90
Complex type is overloaded. The complex type is made of 2 taylors:
TYPE complextaylor
type (taylor) r
type (taylor) i
END TYPE complextaylor
Code that overloads the Taylor packages (including that of Berz) as well as the analysis routines of Lielib
Finally, at the top, this package overloads various useful parameterization of a map: Dragt-Finn, inverse Dragt-Finn, vector fields, vector fields in resonance basis, and, of course, normal forms.
tpsalie_analysis.f90
tpsalie.90
Here operations on maps are overloaded. A new type damap is introduced (as well as other useful types). Map operations are overloaded, for example, concatenation and inverse.
Basic Operations of the TPSA packages involving Taylor series are overloaded here, for example +,-,/, etc... as well as other things such as derivatives. Here we really overload dab.f and newda.f90
Tpsa.f90
definition.f90
Both TPSA Taylor series are merged here. A new type called taylor combines taylorlow and the integer pointer of Berz's TPSA. The complex taylor type and the polymorphic types are also defined there.
Lielib.f
Analysis Routines
newlielib.f90
Berz's TPSA Package: polynomials are represented by integer pointers.
dabnew.f
newda.f90
New TPSA Package
Free standing TPSA packages which could be used on their own. Of course that defeats the purpose of overloading!
define_newda.f90
Defines in new type called taylorlow

Figure 1: Schematic Overview of the Full Polymorphic Package (FPP), down-load from web [7].

not discuss the complex polymorph type but instead concentrate on the real polymorphic type denoted by Real_8 in our package by analogy to REAL*8 in FORTRAN77.

The Real_8 type is defined as follows:

```
TYPE REAL_8
TYPE (TAYLOR) T
DOUBLE PRECISION R
LOGICAL ALLOC
INTEGER KIND
INTEGER I           ! assignment integer
DOUBLE PRECISION S  ! special scaling factor
END TYPE REAL_8
```

A real polymorph is either double precision or taylor. If it is double precision then the integer kind is set to one and the value of the polymorph is contained in the field “r.” If the polymorph is a polynomial then the kind is set to two and the logical alloc is true. This logical is necessary for garbage collection of the polynomials.

If a code is written entirely with types Real_8 rather than true double precisions, and if nothing else is done, then the code will always use the field “r” and the TPSA package will never be invoked. Such a calculation is typically 4 to 5 times slower than a true real floating point calculation as estimated long time ago by Bengtsson. The real power of polymorphism is the ability for a real number to turn Taylor at run time. This is achieved by overloading the equal sign (=). Suppose that the variable A is type real_8: **A=5.d0**. If the kind of A is set to 1, then this line simply assigns 5.d0 to A%r. However if the user has set the kind of A to 0, then this assignment will set polynomial part of A, A%t, to 5.d0 + $s * X_I$. This means that the variable A is now a polynomial in X_I. The quantity A%s is normally defaulted to one unless the user desires otherwise. Of course upon completion of this assignment, the kind of A is set to 2. It is now a polynomial.

To illustrate how these things are done, we will show here the functions that overload addition of two type taylor in TPSA.F90 (polynomials) and the equivalent routine in real_polymorph.f90.

```
FUNCTION ADD( S1, S2 )
 TYPE (TAYLOR) ADD
 TYPE (TAYLOR), INTENT (IN) :: S1, S2
  CALL ASS(ADD)        !assigns add to a global scratch variable
IF(OLD) THEN           ! old=.true. refers to berz's da-package
 CALL DAADD(S1%I,S2%I,ADD%I)
ELSE
 CALL NEWDAADD(S1%J,S2%J,ADD%J)
ENDIF
END FUNCTION ADD
```

This function ADD, in TPSA.F90, overloads the addition of two polynomials by calling the appropriate subroutine of the DA-package. There are routines for all possible operations (adding scalar to polynomials, multiplication etc...) Now let us look at the equivalent function for the polymorph.

```
FUNCTION ADD( S1, S2 )
  TYPE (REAL_8) ADD
  TYPE (REAL_8), INTENT (IN) :: S1, S2
INTEGER LOCALMASTER
SELECT CASE(S1%KIND+3*S2%KIND)
   CASE(4)
   ADD%R=S1%R+S2%R
   ADD%KIND=1
   CASE(5,7,8)
LOCALMASTER=MASTER
     CALL ASS(ADD)
     SELECT CASE(S1%KIND+3*S2%KIND)
      CASE(5)
      ADD%T= S1%T+S2%R
      CASE(7) ;ADD%T= S1%R+S2%T ;CASE(8); ADD%T= S1%T+S2%T
     END SELECT
MASTER=LOCALMASTER
         CASE DEFAULT
WRITE(6,*) " TROUBLE IN ADD "
PAUSE
END SELECT
END FUNCTION ADD
```

Looking at the above function, if S1%kind and S2%kind are both equal to one, then CASE(4) will be selected. The "double precision" field of the polymorphs S1 and S2 will be added and assigned to the double precision field of ADD. If both kinds are 2, then CASE(8) will be selected and the taylor parts will be added. This will invoke the other function ADD contained in TPSA.F90. Of course there are also the cases corresponding to a real added to a taylor series. These are CASE(7) and CASE(8). We mentioned that a polymorph could have kind=0; this should not happen during an operation and therefore the code will reach the DEFAULT case and the program will pause.

The techniques used here could be written more compactly in C++. Also the assignment of global variables is rather involved here: MASTER, LOCALMASTER,etc…are all related to the global scratch variables gymnastics. We will not describe these things here.

This completes our short description of polymorphism. Now we will show some examples of its use in a tracking code.

3 EXAMPLE OF USAGE IN SIXTRACK

```
PROGRAM TEST
USE TRACKING
TYPE(BEAM_LINE) ALS
REAL*8 CLOSED_ORBIT(6)
TYPE(REAL_8) Y(6)
TYPE(DAMAP) A
TYPE(TAYLOR) BETA(2)
TYPE(NORMALFORM) NORMAL_FORM
INTEGER I,ik, ND2
DO I=1,6 ;CLOSED_ORBIT(I)=0.D0 ;ENDDO
CALL read_BEAM_LINE(ALS,'ALS.TXT')
write(6,*) " Number of quadrupoles 0 or 2 "
read(5,*) ik ! If k=0 the quadrupoles are not parameters
IF(IK==0) CALL KILL(ALS) ! MAKES ALL POLYMORPHIC INDEX p%I TO 0
CALL FIND_orbit(ALS,CLOSED_ORBIT,1,nocavity)
if(ik==2) then ! Two families of quadrupoles
 PARA_REMAIN=.TRUE.
 do i=1,als%n
  if(als%mag(i)%name(1:2)=='Q1') als%magp(i)%bn(2)%i=6
  if(als%mag(i)%name(1:2)=='Q2') als%magp(i)%bn(2)%i=7
 enddo
endif
ND2=4
call init(3,ND2/2,ik+1,0,BERZ)
call alloc(y,6);call alloc(NORMAL_FORM)
call alloc(BETA,2);call alloc(A)
y=5
do i=1,6; y(i)=CLOSED_ORBIT(i) ;enddo
CALL TRACK(ALS,Y,1,1,-nocavity)
NORMAL_FORM=y
Y=5
do i=1,ND2;y(i)=NORMAL_FORM%A_t%V(i)+CLOSED_ORBIT(I);enddo
DO I=ND2+1,6;Y(I)=CLOSED_ORBIT(I);ENDDO
do i=1,als%n-1
CALL TRACK(ALS,Y,I,I+1,-NOCAVITY)   !Tracking of NORMAL_FORM%A_t
 A=Y                               !POLYMORPHS MADE INTO A DAMAP
 WRITE(16,*) ALS%MAG(1)%NAME
 BETA(1)=(A%V(1).PAR.'1000')**2+(A%V(1).PAR.'0100')**2
 CALL DAPRINT(BETA(1),16)
 BETA(2)=(A%V(3).PAR.'0010')**2+(A%V(3).PAR.'0001')**2
 CALL DAPRINT(BETA(2),16)
 BETA(1)=BETA(1).D.5 ; CALL DAPRINT(BETA(1),16)
 BETA(2)=BETA(2).D.5 ; CALL DAPRINT(BETA(2),16)
enddo
END PROGRAM TEST
```

In this short program, the one-turn fully coupled map is computed around its fixed point and then normalised. The canonical transformation is contained in the field NORMAL_FORM%A_T. From the theory of normal forms, it is expected that this transformation evolves like the map itself. Therefore it is propagated using the tracking command TRACK(ALS,Y,I,I+1,-NOCAVITY) through element number "I." In this particular example, we tracked the beta functions and their derivatives with respect to delta (fifth variable). The reader notices that in this example the quadrupole strengths are potential runtime parameters.

4 CONCLUSION

The full package is operational and its usefulness has been demonstrated in conjunction with the tracking tools for the LHC, HERA, and ALS lattices. Besides runtime parameter dependence, it also includes stochastic envelopes (for radiation). All calculations use arbitrary vector fields.

5 ACKNOWLEDGEMENTS

One of the authors (E.F.) would like to thank F. Ruggiero for his invitation to join the SL–AP group at CERN for a period of 3 months. He also acknowledges the help of A. Xiao at DESY and the hospitality of F. Willeke.

6 REFERENCES

[1] M. Berz, Part. Accel., 1989, Vol. 24, pp. 109–124.

[2] M. Berz, É. Forest and J. Irwin, Part. Accel., 1989, Vol. 24, pp. 91–107.

[3] É. Forest, LBL differential algebra package and LieLib, unpublished.

[4] M. Berz, "COSY INFINITY 8", NSCL Technical Report MSUCL-1088, Michigan State University, 1998.

[5] V.K. Decyk, C.D. Norton, and B.K. Szymanski, "How to Support Inheritance and Run-Time Polymorphism in Fortran 90", In Computer Physics Communications, Vol. 115, pp. 9-17, 1998.

[6] F. Schmidt, CERN SL/94–56(AP) (1994).

[7] F. Schmidt, "SixTrack web page", http://wwwslap.cern.ch/f̃rs.

[8] Visit the following web sites: http://www-ap.fnal.gov/ michelot/, http://wwwslap.cern.ch/classic/.