Tonto: A Fortran Based Object-Oriented System for Quantum Chemistry and Crystallography

Dylan Jayatilaka and Daniel J. Grimwood

Chemistry, School of Biomedical and Chemical Sciences,
University of Western Australia, Crawley 6009, Australia

Abstract. Tonto is an object oriented system for computational chemistry. This paper focuses mainly on the `Foo`, the object oriented language used to implement Tonto. `Foo` currently translates into Fortran 95. It offers almost all the features of the coming Fortran 2000 except for dynamic types. It goes beyond the Fortran standard in that parameterised types and template-like inheritance mechanisms are provided. Since the method is based on textual inclusion, it generates code which is easy for the compiler and human to understand. Example code is given, and possible future work on the language is discussed.

1 Introduction

Careful studies have demonstrated that over the useful lifetime of a software product, testing, documentation, minor extension, and "bug-fixing" account for the majority of the programming effort [1]. At the same time, the ability to join together different software technologies in computational chemistry is currently extremely poor: at our disposal are monolithic program packages, for example Mopac [2] and Gaussian [3]. These packages are often poorly documented at the code level; many were developed over decades in an ad-hoc manner. Consequently, these packages require experts in order to extend and modify them. Further, the extensions may not be available to all researchers for further development because the source code may be subject to restricted distribution policies.

Over the last five years we have developed a system called Tonto, which attempts to address these problems. Tonto is mainly intended for computational chemists with new ideas, but who are discouraged by the time it might take to understand and modify old, monolithic programs in order to implement those ideas. The three main goals of Tonto are, in order of importance:

1. To provide useful open-source tools for computational chemistry
2. To be simple to use, understand, maintain, and modify
3. To provide code that executes quickly.

Fortran was chosen as the language to implement the Tonto system because of large existing base of Fortran software in computational chemistry, and also because there is a tradition of producing highly optimising Fortran compilers for numerical applications. At the same time, however, it was difficult to ignore the general consensus in the software engineering community that object oriented programming techniques form the best basis for developing well structured and modular code—the kind of structured code that can be easily modified and joined, as required to solve modeling problems in chemistry. Although Fortran 95 is an object oriented language, there are significant deficiencies. That is why it was decided to develop a new dialect of Fortran 95 which more fully supports object orientation. We call the resulting language `Foo`, which stands for object oriented Fortran in reverse (since true polymorphic object orientation is not really possible in Fortran just yet). The `Foo` language forms the central part of the Tonto system.

We are not able to discuss all aspects of the Tonto system in this article. Instead, we focus on a description of the `Foo` language and its features, especially its object oriented features. Some of the object-oriented aspects of `Foo` have been separately implemented in Fortran 90 [4,5,6,7,8,9,10,11,12,13,14], for example, the use of the `self` type [11], template inheritance using textual replacement [13, 14], and the passing of data and functions together [5]. We give some examples of the code written in `Foo`, together with its translation into Fortran 95, and output. Our conclusions and outlook for the system are given at the end.

2 The `Foo` Preprocessor

The `Foo` language is currently implemented as a `perl` script which translates into standard Fortran 95. Since the `Foo` language is almost identical to Fortran at the expression level, it can be viewed as a dialect of Fortran.

The use of preprocessors to implement object oriented languages is not new. For example, the first implementation of C++ was as a preprocessor translating into C [15]. For object oriented styles, preprocessing is particularly useful for incorporating parameterised types into languages that only support the static specification of type parameters [16].

The syntax of the `Foo` language is similar to that of Python, Eiffel and Sather [1]. The most important syntactic element of `Foo` is the use of "dot notation". In this notation, a dot "`.`" expands into the Fortran 95 "`%`" type separator symbol if the symbol after the dot refers to a component of the derived type of the object, otherwise into a procedure call. By using this notation, it is possible for the programmer to "hide" the detailed implementation of a type (and hence a module) from a casual user in the sense that a user need not worry about whether a particular element is a procedure or a derived type component.

There are also clear practical advantages to using the `Foo` language. A simple character count on the existing Tonto library shows that coding in `Foo` eliminates 40% of characters compared to the corresponding Fortran code. We believe that this is a real figure, rather than a reflection of a poor translation into Fortran. It is easy to verify that the emitted Fortran code is clean, efficient and understandable; a programmer could equally work with the Fortran code emitted from the `Foo` preprocessor, as with the `Foo` code itself.

The following is a brief description and discussion of the features of the Foo preprocessor.

- **Dot notation**. Dot notation, as used in most object oriented programming languages, mostly eliminates the need for call statements.
- **HTML and other code documentation facilities**. A translator automatically generates HTML documentation from the source code, which is fully cross-referenced. Syntax highlighting and routine cross-referencing are also provided for the vim editor.
- **Automatic inclusion of use'd modules**. The Foo language automatically generates a list of the modules required by a particular program or module, and the routines needed from those modules. This greatly alleviates the maintenance burden from the programmer, and can reduce the compilation resources for complex modules.
- **Automatic self variable inclusion, and rudimentary closures**. The Foo preprocessor automatically inserts variable self as the first dummy argument for every procedure in a module. The type of self is of the same kind as that described by the module. This automatic mechanism can be suppressed if desired. Such "selfless" procedures are typically passed to other procedures (they may be, for instance, procedures used for numerical integration). Even though the self variable does not appear explicitly as a procedure argument in a selfless procedure, it may be stored temporarily as a module variable (typically called saved_self) at the point of the procedure call, and then passed to a local self variable as the first executable statement within the selfless routine. By this mechanism, the state of the function at the point of call can be stored i.e. this becomes a way to define functions with partially evaluated arguments (such notion is called a "closure", or "lambda expression" in the Lisp or Scheme languages).
- **Overloading and interface files**. Procedures with exactly the same name can be written within a particular module, provided that they can be distinguished by their call signature. The interface file required for this overloading is automatically generated.
- **Multiple template inheritance**. Foo allows a textual inheritance mechanism for modules based on derived types which are built as compositions of other, simpler derived types; and also for modules which are related to one another by a dependence on a type parameter (for example, matrices of real and complex numbers). This mechanism is not restricted to inheriting code from a single module.
- **Automated build procedure**. The build procedure for automatically takes account of the dependencies generated through the inheritance mechanism, and solves the problem of cascading module compilation [17], greatly reducing compilation times.
- **Preconditions, postconditions, and error management facilities**. Preconditions and postconditions can be placed at the beginning and end of procedures to ensure that the minimum requirements for a routine to perform its task correctly are met. This "design-by-contract" style greatly helps

in reducing run-time errors. If an error is detected during program execution, a traceback of the call stack may elucidate where it happened. These features can be disabled or enabled when the program is compiled.

- **Memory management facilities**. Fortran 95 offers dynamic memory allocation method and pointers, but this can lead to memory leaks. `Foo` can automatically check if a procedure is memory conserving, unless specifically marked `leaky`. A running total of allocated memory can be printed out during program execution to trace routines where unintentional memory leaks and sinks occur. This facility can be enabled or disabled when the program is compiled.
- **Well defined input syntax**. The Tonto system defines a (BNF) syntax for input file format, which can be used both for control of program execution and for data interchange. Code tools are provided for this through the `Foo` inheritance mechanism. An automatic mechanism is provided whereby unrecognised input file syntax generates an error message listing the keywords allowed at that point. This greatly facilitates debugging input files when used with the manual.
- **Conditional compilation of `pure` and `elemental` routines**. Fortran 95 disallows write statements within procedures which are `pure` or `elemental`, a great annoyance for testing purposes. `Foo` code can be conditionally compiled with or without these keywords.
- **Automatic `end` keyword completion**. The `end` keyword can be used to stand for any of the usual block termination words: `end program`, `end module`, `end subroutine`, `end function`, `end if`, `end do`, `end interface`, `end select`, `end type`, `end forall`. This encourages the use of indentation. Programmers may choose not to use this feature, and can insert the keywords for themselves.
- **Automatic `subroutine` and `function` detection**. The `Foo` language automatically inserts `subroutine` and `function` keywords, to eliminate typing.
- **Protection against changes in the language elements**. The `Foo` preprocessor represents a minimal approach to object oriented programming, and it is conceivable that it may be translated into a language other than Fortran 95 at a future date. There is also the possibility of translation into more than one language. For example, automatic translation into the Python language may be considered, for "high level" objects.

3 Examples of the Tonto System

3.1 A Short Program Using Some Tonto Modules

Figure 1 presents a sample program written in `Foo`, which makes use of the `TEXTFILE`, `REALMAT`, and `GAUSSIAN2` modules. These modules are concerned with, respectively, text file manipulation, real matrix operations, and pairs of gaussian functions (especially quantum mechanical integrals). The corresponding Fortran 95 source emitted by the `Foo` preprocessor is shown in Figure 2. The output for the program is shown in Figure 3.

```
program run_example

   implicit none

   g :: GAUSSIAN2        ! A GAUSSIAN2 object
   S :: REALMAT*         ! A pointer to a real matrix

   tonto.initialize      ! Tonto system accounting (memory, call stack)
   stdout.create_stdout ! Create the "stdout" file
   stdout.open           ! Open it.
   tonto.set_error_output_file(stdout)

   stdout.flush
   stdout.text("First, define a gaussian pair ...")
   g.set(l_a=0,pos_a=[0.0d0,0.0d0,0.0d0],ex_a=0.3d0, &
         l_b=1,pos_b=[0.0d0,0.0d0,0.1d0],ex_b=0.5d0)
   g.put                     ! Output to "stdout"
   stdout.flush              ! Observe dot notation access to "g"
   stdout.show("The position of the first gaussian is ",g.a.pos)
   stdout.show("The exponent of the first gaussian is ",g.a.ex)

   stdout.flush
   stdout.text("Change the precision of the output to 3 decimal")
   stdout.text("places, and a field width of 8 characters")
   stdout.set_real_precision(3)
   stdout.set_real_width(8)

   stdout.flush
   stdout.text("Analytically evaluate the nuclear attraction")
   stdout.text("integrals for gaussian pair assuming the nucleus")
   stdout.text("of unit charge is at (0,1,0) :- ")
   stdout.flush
   g.make_nuclear_attraction_ints(S,c=[0.0d0,1.0d0,0.0d0])
   stdout.put(S)

   tonto.report
end
```

Fig. 1. An example program written in Foo source code and making use of a number of Tonto modules

The first observation is that the Foo source code is much shorter than the corresponding Fortran 95 code. The main reason for this is the insertion of the use statements. There is also a minor code reduction coming from the fact that call statements are not used. Although this is only a minor issue, the call syntax of Fortran 95 often causes great annoyance for those accustomed to more modern languages. Unfortunately, such stylistic trivialities greatly affect the popularity of a language.

The declaration of variables is also elegant: the rather verbose Fortran 95 type keyword is removed and replaced by a convention of using uppercase macros for the type. The use of macros also hides the distinction between intrinsic and derived types. This is important for in the case that Foo would be translated into a language other than Fortran, where such distinctions do not exist. Pointer attributes are represented by a star character after the type, a feature that should be familiar to C programmers. In Foo, the declaration order for variables and their attributes is reversed compared to Fortran 95. This style follows the philosophy of an assignment operation — except we are assigning a type rather

```
program run_example

   use TYPES_MODULE
   use SYSTEM_MODULE
   use TEXTFILE_MODULE, only: stdin
   use TEXTFILE_MODULE, only: stdout
   use TEXTFILE_MODULE, only: set_real_width_
   use TEXTFILE_MODULE, only: put_
   use TEXTFILE_MODULE, only: text_
   use TEXTFILE_MODULE, only: open_
   use TEXTFILE_MODULE, only: create_stdout_
   use TEXTFILE_MODULE, only: flush_
   use TEXTFILE_MODULE, only: set_real_precision_
   use TEXTFILE_MODULE, only: show_
   use GAUSSIAN2_MODULE, only: put_
   use GAUSSIAN2_MODULE, only: make_nuclear_attraction_ints_
   use GAUSSIAN2_MODULE, only: set_
   use GAUSSIAN_MODULE, only: n_comp_
   use REALMAT_MODULE, only: create_

   implicit none

   type(gaussian2_type) :: g ! A type(gaussian2_type) object
   real(8), dimension(:,:), pointer :: S ! A pointer to a real matrix

   call initialize_(tonto)      ! Tonto system accounting (memory, call stack)
   call create_stdout_(stdout)  ! Create the "stdout" file
   call open_(stdout)           ! Open it.
   call set_error_output_file_(tonto,stdout)

   call flush_(stdout)
   call text_(stdout,"First, define a gaussian pair ...")
   call set_(g,l_a=0,pos_a=(/0.0d0,0.0d0,0.0d0/),ex_a=0.3d0, &
         l_b=1,pos_b=(/0.0d0,0.0d0,0.1d0/),ex_b=0.5d0)
   call put_(g)                     ! Output to "stdout"
   call flush_(stdout)              ! Observe dot notation access to "g"
   call show_(stdout,"The position of the first gaussian is ",g%a%pos)
   call show_(stdout,"The exponent of the first gaussian is ",g%a%ex)

   call flush_(stdout)
   call text_(stdout,"Change the precision of the output to 3 decimal")
   call text_(stdout,"places, and a field width of 8 characters")
   call set_real_precision_(stdout,3)
   call set_real_width_(stdout,8)

   call flush_(stdout)
   call text_(stdout,"Analytically evaluate the nuclear attraction")
   call text_(stdout,"integrals for gaussian pair assuming the nucleus")
   call text_(stdout,"of unit charge is at (0,1,0) :- ")
   call flush_(stdout)
   call make_nuclear_attraction_ints_(g,S,c=(/0.0d0,1.0d0,0.0d0/))
   call put_(stdout,S)

   call report_(tonto)
end program
```

Fig. 2. The Fortran 95 source generated by the Foo preprocessor, corresponding to the source code in Figure 1

than a value. Procedures in Foo have a similar attribute declaration style (see Figure 4).

Note that in the Fortran 95 source, every routine has an underscore character appended relative to the corresponding Foo routine. These underscored rou-

```
First, define a gaussian pair ...

GAUSSIAN2:

l_a   =          0
l_b   =          1
pos_a =   0.000000   0.000000   0.000000
pos_b =   0.000000   0.000000   0.100000
ex_a  =   0.300000
ex_b  =   0.500000

The position of the first gaussian is    0.000000   0.000000   0.000000
The exponent of the first gaussian is    0.300000

Change the precision of the output to 3 decimal
places, and a field width of 8 characters

Analytically evaluate the nuclear attraction
integrals for gaussian pair assuming the nucleus
of unit charge is at (0,1,0) :-

                1       2       3

        1   0.000   1.651  -0.334

SYSTEM: Memory usage report:

Memory used                =       110 Words
Maximum memory used        =    390718 Words
Memory blocks used         =         3
Maximum memory blocks used =        14
Call stack level           =         0
Maximum call stack depth   =        10
```

Fig. 3. Output corresponding to the programs in Figures 1 and 2

tines have been overloaded. Generic Fortran 95 function interfaces (not shown) have been generated automatically by `Foo` when compiling the modules. Unfortunately, this method generates a very large number of generic names, and compilers have great difficulties to deal with the resulting name space. `Foo` can use another method to generate the generic names: it can prepend each routine name with the name of the module it belongs to. This greatly reduces the number of generic names which are the same, and consequently reduces compilation requirements. Unfortunately, the names generated by this method are often long and exceed the standard length of 31 characters. This restriction will be alleviated in the Fortran 2000 standard when 63 characters. Some compilers already allow long character names.

Note how the dot notation operates: if the `Foo` code refers to a routine, then the corresponding code expands into a procedure call, but if the dot refers to a type component, the corresponding Fortran code uses a % symbol. From the `Foo` programmer's perspective, there is no difference between the two. Thus, the internal type structure of the module can be changed without affecting the code which uses it: functions can replace type components and vice-versa. (Of course, in high performance applications, one must be careful about unnecessary use of function calls, so although the calling code will work, it will probably need to be rewritten for fast execution).

Memory management is dealt with largely automatically by the system. The `allocate` and `deallocate` routines are replaced in the Tonto system by `create` and `destroy` routines. The amount of memory, and which routines are called by the system, are stored in the `tonto` system variable. A printout of the call stack is available at any point in the program (provided that support for this feature was compiled into it, as an option). In principle, profiling features could also be incorporated, but we have not yet done so. At the end of the program, we print out details of the statistics of memory use.

The style of the code is also very important – perhaps as important as the use of object based techniques. Long descriptive names are used (up to 31 characters) and underscores are used liberally to make reading easier. Although typing these names can be annoying, there is a significant payoff in terms of clarity. The whole Tonto system is written this way.

```
append(value) ::: leaky
! Expands self and appends the single scalar "value" onto the end.
  self :: PTR
  value :: ELEMENT_TYPE, IN
  dim :: INT
  if (.destroyed) then; dim = 0
  else;                 dim = .dim
  end
  .expand(dim+1)
  self(dim+1) = value
end
```

Fig. 4. `Foo` source for appending a scalar value to an array, appearing in the `INTRINSICVEC` module. Notice the type parameter `ELEMENT_TYPE`

```
append(value) ::: leaky, get_from(INTRINSICVEC)
! Expands self and appends the single scalar "value" onto the end.
  self :: PTR
  value :: ELEMENT_TYPE, IN
end
```

Fig. 5. Inherited `Foo` source for appending a string value to a string array, appearing in the `STRVEC` module. `ELEMENT_TYPE` has the value `STR` in this module

3.2 A Short Example of Template Inheritance

Figure 4 shows a code fragment for appending an element to a vector, and makes use of the predefined type parameter `ELEMENT_TYPE`. This code appears in the `INTRINSICVEC` "virtual" module – a module which is not compiled, but comprises routines that are "inherited" by other modules. Such modules are called "virtual classes", "abstract types" , or "template classes" in other languages. Figure 5 shows how the inclusion of this code is effected in the `STRVEC` module, a module concerned with one dimensional arrays of strings. In this module, `ELEMENT_TYPE` is replaced by the type `STR`, which is the type corresponding to an element of the array of type `STRVEC`. Notice the use of the `get_from` procedure attribute in `Foo`. The `leaky` attribute indicates that a memory leak occurs in the routine (a warning message will not be generated in this case). Code is inherited only if the routine signatures are identical. Notice also that the body of the routine in the `STRVEC` module is absent. The source code from `INTRINSICVEC` is inserted verbatim.

4 Concluding Remarks

There are, of course, a number of ways to improve the `Foo` language. The syntax of the parameterised type and template inheritance mechanism could be refined. We have also noted that with very large modules, compile times can be very long. Clearly, some syntax for *submodules* needs to be defined, such as has been recommended by the Fortran standards body [18]. The preprocessor code itself is a rather ad-hoc `perl` script: a proper parser-generator should probably be written, perhaps using the `RecDescent` package [19]. Due to the similarity of `Foo` and Fortran, we have not attempted a full BNF syntax specification yet, but this should be done.

Despite these problems, we believe the `Foo` language offers a seamless way to take almost full advantage of the latest Fortran 2000 standard, and features beyond it, without waiting for vendors to produce compilers that adequately support the new standard — a development that may be 5 years or more away.

References

1. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, Inc., second edition, 1997.
2. J. J. P. Stewart. *MOPAC 2002.* Fujitsu Limited, Tokyo, Japan, 1999.
3. Æ. Frisch and M. J. Frisch. *Gaussian 98 User's Reference.* Gaussian, Inc., Pittsburgh PA, second edition, 1999.
4. B. J. Dupee. Object oriented methods using fortran 90. *ACM Fortran Forum*, 13:21–30, 1994.
5. R. Lutowski. Object-oriented software development with traditional languages. *ACM Fortran Forum*, 14:13–18, 1995.
6. C. D. Norton, V. K. Decyk, and B. K. Szymanski. On parallel object oriented programming in fortran 90. *ACM SIGAPP Applied Computing Review*, 4:27–31, 1996.
7. C. D. Norton, V. K. Decyk, and B. K. Szymanski. High performance object-oriented programming in fortran 90. In *Proc. Eighth SIAM Conference on Parallel Processsing for Scientific Computing*, March 1997.

8. J. R. Cary, S. G. Shasharina, C. Cummings, J. V. W. Reynders, and P. J. Hinker. Comparison of c++ and fortran 90 for object-oriented scientific programming. *Computer Phys. Comm.*, 105:20–36, 1997.
9. V. K. Decyk, C. D. Norton, and B. K Szymanski. Expressing object-oriented concepts in fortran 90. *ACM Fortran Forum*, 16:13–18, 1997.
10. L. Machiels and M. O. Deville. Fortran 90: An entry to object-oriented programming for the solution of partial differential equations. *ACM Transactions on Mathematical Software*, 23:32–49, 1997.
11. M. G. Gray and R. M. Roberts. Object-based programming in fortran 90. *Computers in physics*, 11:355–361, 1997.
12. V. K. Decyk, C. D. Norton, and B. K. Szymanski. How to support inheritance and run-time polymorphism in fortran 90. *Computer Phys. Comm.*, 115:9–17, 1998.
13. V. Snyder. Constructive uses for include. *ACM Fortran Forum*, 20:2–4, 2001.
14. A. Markus. Generic programming in fortran 90. *ACM Fortran Forum*, 20:20–23, 2001.
15. B. Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.
16. S. Khoshafian and R. Abnous. *Object Orientation.* John Wiley & Sons, Inc, second edition, 1995.
17. T. Stern and D. Grimwood. Cascade compilation revisited. *ACM Fortran Forum*, 21:12–24, 2002.
18. JTC1/SC22/WG5. Enhanced module facilities in fortran. Technical Report 19767, ISO/IEC, 2002.
19. D. M. Conway. The man(1) of descent. *The Perl J.*, 3:46–58, 1998.