6-18-2012

Fake run-time selection of template arguments in C++

Daniel Langr
Czech Technical University in Prague

Pavel Tvrdík
Czech Technical University in Prague

Tomáš Dytrych
Louisiana State University

Jerry P. Draayer
Louisiana State University

Recommended Citation

Langr, D., Tvrdík, P., Dytrych, T., & Draayer, J. (2012). Fake run-time selection of template arguments in C++. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7304 LNCS*, 140-154. https://doi.org/10.1007/978-3-642-30561-0_11

Fake Run-Time Selection of Template Arguments in C++

Daniel Langr* and Pavel Tvrdík
Czech Technical University in Prague
Department of Computer Systems
Faculty of Information Technology
Thákurova 9, 160 00, Praha, Czech Republic

Tomáš Dytrych and Jerry P. Draayer
Louisiana State University
Department of Physics and Astronomy
Baton Rouge, LA 70803, USA

Abstract

C++ does not support run-time resolution of template type arguments. To circumvent this restriction, we can instantiate a template for all possible combinations of type arguments at compile time and then select the proper instance at run time by evaluation of some provided conditions. However, for templates with multiple type parameters such a solution may easily result in a branching code bloat. We present a template metaprogramming algorithm called `for_id` that allows the user to select the proper template instance at run time with theoretical minimum sustained complexity of the branching code.

Keywords: C++, run-time selection, template arguments, template metaprogramming, type sequences

1 Introduction

C++ templates allow to define a parametrized piece of code for which data types are specified later as template arguments. According to the C++ Standard [ISO03], template arguments must be known at compile time. There are, however, situations where we might want to postpone the choice of template arguments to run time. Consider, e.g., the following ones:

Run-time choice of floating-point precision: Many pieces of today's scientific and engineering software allow programmers to choose the floating-point precision at compile time [Ope11, BBB$^+$10, GJ$^+$10, HW03, VDY05]. If we then want to alternate single-precision and double-precision computations, we need either to recompile programs frequently or to maintain both versions simultaneously.

Minimization of memory requirements: Indexes pointing into arrays of different sizes constitute essential parts of data structures in scientific and engineering software. Let us have an array of size ξ whose elements are indexed from 0 to $\xi - 1$. The minimum number of bits of the unsigned integer data type that is capable to index such an array on a 64-bit computer is then

$$b(\xi) = \min\{\eta \in \{8, 16, 32, 64\} : \xi \le 2^\eta\}. \quad (1)$$

In some software, such as PETSc [BBB$^+$10], users can choose between 32-bit and 64-bit data types for indexes. However, the choice has to be done at compile time and the same data type is then used for all indexes independently of the actual size of indexed arrays.

Reading data from binary files: The program might not know which data types to use until it opens the file at run time. For instance, when reading files based on the HDF5 file format [Theb], we can find out information about the types of stored data sets in the form of numerical constants.

Let us now define the problem we want to address.

Problem 1. Assumptions:

*E-mail: langrd@fit.cvut.cz

1. Suppose `f` is a function object[1] with a templated function call operator. We will call the number of its template parameters the *dimension* of the problem and denote it by d.
2. Let us have d finite sequences of data types $\mathcal{T}_1, \dots, \mathcal{T}_d$, where
$$\mathcal{T}_i = \{t_i^k : k = 1, \dots, n_i\}.$$
3. Let us further have d sequences of mutually exclusive Boolean conditions $\mathcal{C}_1, \dots, \mathcal{C}_d$, where
$$\mathcal{C}_i = \{c_i^k : k = 1, \dots, n_i - 1\},$$
that cannot be evaluated until run time.

We want to *apply* the function object `f`, that is, to call

```
f.operator()<$t_1$, ... , $t_d$>();
```

where
$$t_i = \begin{cases} t_i^k & \text{if there exists } k \in \{1, \dots, n_i - 1\} \text{ such that } c_i^k \text{ is true,} \\ t_i^{n_i} & \text{otherwise.} \end{cases}$$
□

A simple one-dimensional example of Problem 1 is the run-time selection of the floating point precision for some algorithm via a program's command line option.

Example 1. Consider a function object called `algorithm` defined as follows:

```
struct {
    template <typename T>
    void operator()() {
        ... // some code
    }
} algorithm;
```

Let $\mathcal{T}_1 = \{$`float`$,$`double`$\}$ and $\mathcal{C}_1 = \{$`strcmp(argv[1], "1")`$\}$.

□

That is, we want to invoke `algorithm.operator()<double>()` if the value of the first command line options is `"1"`, and `algorithm.operator()<float>()` otherwise. The obvious solution is to branch the code according to the provided condition as follows:

```
if (strcmp(argv[1], "1"))
    algorithm.operator()<float>();
else
    algorithm.operator()<double>();
```

Generally, for Problem 1, we may define the solution based on code branching as follows.

Solution 1.

```
if ($c_1^1$ && $c_2^1$ && $\cdots$ && $c_d^1$)
    f.operator()<$t_1^1$, $t_2^1$, ... , $t_d^1$>();
else if ($c_1^1$ && $c_2^1$ && $\cdots$ && $c_d^2$)
    f.operator()<$t_1^1$, $t_2^1$ , ... , $t_d^2$>();
...
else
    f.operator()<$t_1^{n_1}$, $t_2^{n_2}$, ... , $t_d^{n_d}$>();
```

□

[1] A *function object*, or simply a *functor*, is an object of a class that overloads the *function call operator*. See, for example, Prata [Pra01] or Stroustrup [Str00] for more details.

The drawback of this solution is obvious—the complexity of the branching code grows combinatorially (recall that we need the templated function call operator to be instantiated for all possible combinations of data types). That is, it yields the total number of branches $\mathcal{N}_\Pi = \prod_{i=1}^{d} n_i$.

The lower bound on the number of branches is $\mathcal{N}_\Sigma = \sum_{i=1}^{d} n_i$, because all conditions must be evaluated in the worst case scenario. This lower bound could be achieved by the following *imaginary* code:

```
???? t_1;
if (c_1^1)
    t_1 = t_1^1;
else if (c_1^2)
    t_1 = t_1^2;
...
else if (c_1^{n_1-1})
    t_1 = t_1^{n_1-1};
else
    t_1 = t_1^{n_1};
... // similarly for the remaining dimensions
f.operator()<t_1, t_2, ... , t_d>();
```

Unfortunately, such a code is not valid, because C++ does not allow to assign types[2].

In this paper, we present a solution of Problem 1 that achieves the lower bound on the number of required code branches $\mathcal{N}_\Sigma$. Its application to Example 1 may look simply like:

```
typedef boost::mpl::vector<float, double> fp_types;
...
int fp_id = (strcmp(argv[1], "1")) ? 0 : 1;
for_id<fp_types>(algorithm, fp_id);
```

This solution is based on template metaprogramming [Ale01, BN94, Gen11, VJ02] and sequences of types from Boost Metaprogramming Library (MPL) [AG04]. Although we cannot choose template arguments at run time, we can choose positions (ids) of desired data types inside type sequences. Based on these ids, the presented `for_id` algorithm invokes the desired template instance[3]. Since both approaches are syntactically similar, we refer to our solution as *fake* run-time selection of template arguments.

The rest of the paper is organized as follows. In Section 2, the previous work related to our problem is presented and analyzed. Section 3 covers design and implementation of the proposed `for_id` algorithm. In Section 4, experiments are described and their results are presented and discussed. Section 5 summarizes the properties of the `for_id` algorithm and describes its usage in an existing high performance computing (HPC) code.

Note that this paper is an extended version of our previous work [LTDD12]. The additional material includes the following items:

- We tested the compatibility of the `for_id` algorithm with various C++ compilers. The results of these experiments are presented and discussed in Section 4.
- We measured the dependence of compilation time and memory requirements of the `for_id` algorithm on the problem dimension and on the length of type sequences. The results of these experiments are presented and discussed in Section 4.
- Section 2 was extended to include additional details about the related work.
- Due to the space limitations, the syntax of the code excerpts is heavily compressed in [LTDD12]. In effect, the code presented therein requires modifications to become valid C++ code. In this paper, valid C++ code, which can be easily embedded into real programs, is shown.

[2]Using `typedef`s instead of assignment would not help here, because even though most C++ compilers do accept `typedef`s in a function body, such type definitions would not propagate from inside the code branches.

[3]In fact, `for_id` is an ordinary C++ function template (not a metafunction). However, its functionality matches the category that is called *algorithms* in Boost MPL.

2 Related Work

C++ templates and template metaprogramming have always been intended to be utilized primarily at compile time. Boost MPL [AG04] is a widely-used general-purpose metaprogramming library advertised as *"high-level C++ template metaprogramming framework of compile-time algorithms, sequences and metafunctions"* [Thea]. There are many compile-time algorithms in Boost MPL, but only one run-time algorithm: `for_each`. The call `boost::mpl::for_each<seq>(f)` applies the function object `f` (calls its function call operator) to every element of the type sequence `seq` at run time. There are two significant differences between Boost MPL's `for_each` and our `for_id`:

1. `for_each` applies the function object to all elements in the type sequence. `for_id` applies the function object to a single element only—the one that is identified by its position (id).
2. `for_each` is one-dimensional, that is, it can operate only on a single type sequence. `for_id` is multi-dimensional and we designed it with no imposed limit of the number of dimensions (type sequences). This limit is given solely by the compiler.

Generic Image Library (GIL) [BJ] allows to design generic algorithms for different types of images that are not known until run time. According to the actual image properties, such as the color space or bit depth, a proper template instance of the algorithm is invoked at run time. However, this functionality is tightly coupled with GIL and it is not presented as an independent metaprogramming algorithm for general-purpose usage. Some implementation details are described in the paper of Bourdev and Järvi [BJ11], but for deeper understanding of their solution, we need to study undocumented functions and class templates from GIL's source code[4].

3 Design and Implementation

3.1 Notation

We adhere to the following notation rules in the text below:

1. The header files that should be included to compile our examples are listed in Appendix A.
2. We suppose that the `using` directive is provided for the `boost::mpl` namespace, that is:

```
using namespace boost::mpl;
```

3. We use the τ symbol for MPL iterators such that `deref<`τ_i^k`>::type` is equal to t_i^k, and $\tau_i^{n_i+1}$ denotes `end<`$\mathcal{T}_i$`>`.

By the symbol *id* we denote a zero-based index into a type sequence. We say that *id* is *valid* for the sequence `S` if it belongs to $\{0, \dots, $`size<S>::value`$ - 1\}$.

3.2 Initial Step

Let us first define a metafunction `pos` that returns a zero-based index of a type within a type sequence (that is, `pos <`$\mathcal{T}_i, t_i^k$`>` is equal to $k - 1$).

```
template <typename S, typename T>
struct pos : distance<
    typename begin<S>::type,
    typename find<S,T>::type
>::type { };
```

[4]Looking at the `apply_operation_base.hpp` header file, we find that proper template instances are selected via extensive `switch` statements in the `apply` member function of the `apply_operation_fwd_fn` class template. To cover numerous possibilities that may arise, this class template is multiple times partially specialized. Direct definition of these specializations would lead to an extremely large header file, thus, the Boost Preprocessor metaprogramming library is utilized here. The size of the `apply_operation_base.hpp` header file is only 10.5 kB, however, after removing all other included header files, we measured that the size of its preprocessed output is 2.7 MB.

Our initial solution of the one-dimensional Problem 1 is then:

```
1  // primary template
2  template <
3     typename S1,
4     typename B1 = typename begin<S1>::type,
5     typename E1 = typename end<S1>::type
6  >
7  struct for_id_impl_1 {
8     template <T>
9     static void execute(T& f, int id1) {
10       if (pos<S1, typename deref<B1>::type>::value == id1)
11          f.template operator()<typename deref<B1>::type>();
12       else if (1 == distance<B1, E1>::value)
13          throw std::invalid_argument("");
14       else
15          for_id_impl_1<S1, typename next<B1>::type, E1>::execute(f, id1);
16    }
17 };
18
19 // partial specialization
20 template <typename S1, typename E1>
21 struct for_id_impl_1<S1, E1, E1> {
22    template <T> static void execute(T& f, int id1) { }
23 };
```

It iterates over the type sequence `S1` either until the position of the actual type matches the desired *id*, or until the end of the sequence is reached. In the former case, the function object is applied. In the latter case, an exception is thrown. The partial specialization defined at lines 20–23 is never reached at run time, however, it is needed to stop the recursive instantiation at compile time.

Let us go back to our Example 1 where $\mathcal{T}_1 = \{$`float`, `double`$\}$. What happens when we now call `for_id_impl_1<`$\mathcal{T}_1$`>::execute(algorithm, id1)` and `id1` is 1?

1. At lines 4–5, the default arguments are resolved, resulting in `<`$\mathcal{T}_1, \tau_1^1, \tau_1^3$`>`.

2. The `execute` function, defined at lines 8–16, is invoked, wherein `id1` is equal to 1 and `pos<`$\mathcal{T}_1, t_1^1$`>::value` is equal to 0.

3. The condition at line 10 is hence not satisfied. At the same time, the condition at code line 12 is not satisfied either, because `distance<`τ_1^1, τ_1^3`>::value` is 2. Hence, the following command is executed at line 15:
   `for_id_impl_1<`$\mathcal{T}_1, \tau_1^2, \tau_1^3$`>::execute(algorithm, 1)`.

4. The condition at line 10 is now satisfied, because both `pos<`$\mathcal{T}_1, t_1^2$`>::value` and `id1` are equal to 1. Since `deref<`τ_1^2`>::type` is equal to t_1^2 which is `double`, the following command is executed at line 11:
   `algorithm.template operator()<double>()`.

This is exactly what we wanted, i.e., to select the `<double>` instance of the function call operator of `algorithm` by a run-time parameter `id1` (zero-based index of `double` in $\mathcal{T}_1$ is 1).

What would happen if `id1` would be invalid—for example, if it would be equal to 10? Up to the point 3 in the previous list, the behavior would be the same. However, then it would run differently:

4′. The condition at code line 10 is not satisfied, because `id1` is equal to 10 and `pos<`$\mathcal{T}_1, t_1^2$`>::value` is equal to 1. However, the condition at line 12 is now satisfied, since `distance<`τ_1^2, τ_1^3`>::value` is 1. We are already at the end of $\mathcal{T}_1$ and there are no more types to iterate over. Hence, the exception that indicates the wrong `id1` argument is thrown.

3.3 Extension to Multiple Dimensions

The following solution for two dimensions is based on the same idea of iterating over type sequences—we just have two of them and for each one a separate *id*.

```
 1  // primary template
 2  template <
 3     typename S1,
 4     typename S2,
 5     typename B1 = typename begin<S1>::type,
 6     typename B2 = typename begin<S2>::type,
 7     typename E1 = typename end<S1>::type,
 8     typename E2 = typename end<S2>::type,
 9     typename T1 = typename deref<B1::type>
10  >
11  struct for_id_impl_2 {
12     template <typename T>
13     static void execute(T& f, int id1, int id2) {
14        if (pos<S1, deref<B1>::type>::value == id1)
15           for_id_impl_2<
16              S1, S2, E1, B2, E1, E2, typename deref<B1>::type
17           >::execute(f, id1, id2);
18        else if (1 == distance<B1, E1>::value)
19           throw std::invalid_argument("");
20        else
21           for_id_impl_2<
22              S1, S2, typename next<B1>::type, B2, E1, E2, T1
23           >::execute(f, id1, id2);
24     }
25  };
26
27  // partial specialization #1
28  template <
29     typename S1, typename S2, typename B2,
30     typename E1, typename E2, typename T1
31  >
32  struct for_id_impl_2<S1, S2, E1, B2, E1, E2, T1> {
33     template <typename T>
34     static void execute(T& f, int id1, int id2) {
35        if (pos<S2, typename deref<B2>::type>::value == id2)
36           f.template operator()<T1, typename deref<B2>::type>();
37        else if (1 == distance<B2, E2>::value)
38           throw std::invalid_argument("");
39        else
40           for_id_impl_2<
41              S1, S2, E1, typename next<B2>::type, E1, E2, T1
42           >::execute(f, id1, id2);
43     }
44  };
45
46  // partial specialization #2
47  template <
48     typename S1, typename S2, typename E1, typename E2, typename T1
49  >
50  struct for_id_impl_2<S1, S2, E1, E2, E1, E2, T1> {
51     template <typename T>
52     static void execute(T& f, int id1, int id2) { }
53  };
```

In the primary template defined at lines 2–25, the program iterates over the first type sequence `S1`. When the desired type is found, that is, when the condition at line 14 is satisfied, the function object cannot be applied, because the second type is not yet known. The resolved type is stored into the template parameter `T1` and the process proceeds to the second dimension. This is done by setting `B1` to `E1`, which causes the transition to the first partial specialization defined at lines 28–44. This partial specialization iterates over the second type sequence and when `id2` is matched at line 35, the function object can be applied, since all data types are now known.

Extension to 3 and more dimensions can be done by following the same pattern. However, this approach has a quadratic complexity of the number of definitions. For the dimension d, we need $d + 1$ definitions—a primary template and d partial specializations. So, if we want to support all dimensions from 1 to some $d_{\max}$, we finally need $d_{\max}(d_{\max} + 3)/2 = O(d_{\max}^2)$ definitions, which is not optimal.

3.4 The Optimal Solution

We present here a solution that needs only $2d_{\max} + 1 = O(d_{\max})$ definitions. It primarily uses only $d_{\max} + 1$ definitions that are common for all $d \in \{1, \ldots, d_{\max}\}$. For $d_{\max} = 2$ these definitions are the following:

```
// primary template
template <
   int D,
   typename S1,
   typename S2 = vector<>,
   typename B1 = typename begin<S1>::type,
   typename B2 = typename begin<S2>::type,
   typename E1 = typename end<S1>::type,
   typename E2 = typename end<S2>::type,
   typename T1 = typename deref<B1>::type,
   typename T2 = typename deref<B2>::type
>
struct for_id_impl {
   template <typename T>
   static void execute(T& f, int id1, int id2 = 0) {
      if (pos<S1, typename deref<B1>::type>::value == id1)
         if (1 == D)
            executor<D, typename deref<B1>::type, T2>::execute(f);
            // f.template operator()<typename deref<B1>::type>();
         else
            for_id_impl<
               D, S1, S2, E1, B2, E1, E2, typename deref<B1>::type
            >::execute(f, id1, id2);
      else if (1 == distance<B1, E1>::value)
         throw std::invalid_argument("");
      else
         for_id_impl<
            D, S1, S2, typename next<B1>::type, B2, E1, E2, T1
         >::execute(f,id1);
   }
};

// partial specialization #1
template <
   int D, typename S1, typename S2, typename B2,
   typename E1, typename E2, typename T1, typename T2
>
struct for_id_impl<D, S1, S2, E1, B2, E1, E2, T1, T2> {
   template <typename T>
   static void execute(T& f, int id1, int id2 = 0) {
      if (pos<S2, typename deref<B2>::type>::value == id2)
         executor<D, T1, typename deref<B2>::type>::execute(f);
         // f.template operator()<T1, typename deref<B2>::type>();
      else if (1 == distance<B2, E2>::value)
         throw std::invalid_argument("");
      else
         for_id_impl<
            D, S1, S2, E1, typename next<B2>::type, E1, E2, T1
         >::execute(f,id1,id2);
   }
};

// partial specialization #2
template <
```

```
55     int D, typename S1, typename S2,
56     typename E1, typename E2, typename T1, typename T2>
57  struct for_id_impl<D, S1, S2, E1, E2, E1, E2, T1, T2> {
58     template <typename T>
59     static void execute(T& f, int id1, int id2 = 0) { }
60  };
```

The idea of iterating over type sequences and moving to the next dimension after resolving the actual one is preserved. Comparing `for_id_impl` with the previously defined template `for_id_impl_2`, we find the following essential differences:

1. The `D` template parameter, equal to the number of dimensions d, was introduced.
2. The `S2` template parameter and the `id2` function parameter have default values, because they are useless for one-dimensional problems and we do not want to force the user to specify meaningless values for them.
3. The condition at line 17 was introduced, because when the type is resolved for a particular dimension, we need to select the further action according to the number of dimensions of the problem. At line 17, where the first type is already known, we need either

   (a) to apply the function object for one-dimensional problems (`D` is 1),

   (b) or to move to the next dimension for two-dimensional (generally more-than-one-dimensional) problems.

   As we further see, no such condition is needed for the first partial specialization, because the `executor` structure is defined only for `D` being equal to 1 or 2.
4. Unfortunately, within this new solution, we cannot apply the function object directly inside `for_id_impl::execute`, as is suggested by the comments at lines 19 and 43. The reason is that for a two-dimensional problem, we suppose a function object with a templated function call operator that has exactly two template parameters. However, in such a case, the call
   `f.template operator()<typename deref<B1>::type>()`
   at line 19 would trigger a compilation error, because no such one-parameter version of the function call operator exists. We have solved this problem by delegation of the application of the function object to a helper structure called `executor` that is defined as follows:

   ```
   template <int D, typename T1, typename T2>
   struct executor;

   template <typename T1, typename T2>
   struct executor<1, T1, T2> {
      template <typename T>
      static void execute(T& f) {
         f.template operator()<T1>();
      }
   };

   template <typename T1, typename T2>
   struct executor<2, T1, T2> {
      template <typename T>
      static void execute(T& f) {
         f.template operator()<T1, T2>();
      }
   };
   ```

As in Section 3.3, the extension to 3 and more dimensions is straightforward. For each supported dimension, we need to define one specialization of `for_id_impl` and one of `executor`. Hence, we need $2d_{\max} + 1$ definitions in total.

It might seem that this new solution introduces some overhead when compared with the one in Sections 3.2 and 3.3, because there is too much code branching. However, we need to realize that the conditions at lines 17, 24 and 44 may be evaluated at compile time and an efficient compiler will not propagate the branching into the resulting machine code.

3.5 Wrapping Up

Although `for_id_impl` already solves Problem 1, we can make things more comfortable by introducing the following wrappers:

```
// one-dimensional case
template <typename S1, typename T>
void for_id(T& f, int id1) {
  for_id_impl<1, S1>::execute(f, id1);
}

// two-dimensional case
template <typename S1, typename S2, typename T>
void for_id(T& f, int id1, int id2) {
  for_id_impl<2, S1, S2>::execute(f, id1, id2);
}
```

which allows to write simply `for_id<seq>(f, id)` instead of
`for_id_impl<1, seq>::execute(f, id)`.

3.6 Summary

With `for_id`, we may write the solution of Problem 1 as follows.

Solution 2.

```
int $id_1$;

if ($c_1^1$)
    $id_1$ = 0;
else if ($c_1^2$)
    $id_1$ = 1;
...
else if ($c_1^{n_1-1}$)
    $id_1$ = $n_1$ - 2;
else
    $id_1$ = $n_1$ - 1;

... // similarly for the remaining dimensions

for_id<$\mathcal{T}_1$, $\mathcal{T}_2$, ... , $\mathcal{T}_d$>(f, $id_1$, $id_2$, ... , $id_d$);
```

□

Hence, this solution achieves the minimal number of code branches $\mathcal{N}_\Sigma$.

4 Experimental Results

4.1 Test Program

To evaluate `for_id`, we have developed a program for computing the dominant eigenvalue of a real symmetric matrix that is obtained from a file based on the Matrix Market file format [BPR96]. The file name is specified as a program's command line option, therefore, the number of matrix rows (columns) and the number of nonzero elements are not known until run time. Within the program, the matrix is stored in the memory in the *coordinate storage sparse format* using the following data structure:

```
struct Matrix {
  uint64_t n, z;
  void *i, *j, *a;
} m;
```

where

- `n` is the number of matrix rows;
- `z` is the number of matrix nonzero elements;

- `i`, `j` and `a` are arrays containing *row indexes*, *column indexes*, and *values* of matrix nonzero elements, respectively.

The program looks like:

```cpp
#if defined CASE_FOR_ID
typedef vector<float, double> fp_types;
typedef vector<uint8_t, uint16_t, uint32_t, uint64_t> ind_types;
#endif

int main(int argc, char* argv[]) {
   std::ifstream ifs(argv[1]);
   while ('%' == ifs.peek())
      ifs.ignore(1024, '\n');
   ifs >> m.n >> m.n >> m.z;
   uint64_t q = boost::lexical_cast<uint64_t>(argv[2]);

   MatrixReader mr(m, ifs);

#if defined CASE_F_16
   mr.operator()<float, uint16_t>();
#elif defined CASE_F_32
   mr.operator()<float, uint32_t>();
#elif defined CASE_D_16
   mr.operator()<double, uint16_t>();
#elif defined CASE_D_32
   mr.operator()<double, uint32_t>();
#elif defined CASE_FOR_ID
   int fp_id = (strcmp(argv[3], "1")) ? 0 : 1;

   int ind_id = 3;
   if (m.n <= (1UL << 8))
      ind_id = 0;
   else if (m.n <= (1UL << 16))
      ind_id = 1;
   else if (m.n <= (1UL << 32))
      ind_id = 2;

   for_id<fp_types, ind_types>(mr, fp_id, ind_id);
#else
   #error No program case selected!
#endif

double lambda;
PowerMethod pm(m, q, lambda);

#if defined CASE_F_16
   pm.operator()<float, uint16_t>();
#elif defined CASE_F_32
   pm.operator()<float, uint32_t>();
#elif defined CASE_D_16
   pm.operator()<double, uint16_t>();
#elif defined CASE_D_32
   pm.operator()<double, uint32_t>();
#elif defined CASE_FOR_ID
   for_id<fp_types, ind_types>(pm, fp_id, ind_id);
#endif

  std::cout << "Lambda: " << lambda << "\n";

  return 0;
}
```

It consists of the following steps:

Table 1: Cases of the test program.

Case	Data type	
	floating point	indexing
C^f_{16}	`float`	`uint16_t`
C^f_{32}	`float`	`uint32_t`
C^d_{16}	`double`	`uint16_t`
C^d_{32}	`double`	`uint32_t`
C^*_*	resolved by `for_id`	resolved by `for_id`

1. The file input stream `ifs` for the matrix file is opened at line 7.
2. The header and comments are skipped at lines 8–9.
3. The number of rows/columns and the number of nonzero elements are read at line 10.
4. The number of iterations for the power method is obtained from the second command line option at line 11.
5. At line 13, the `mr` function object is defined. It is responsible for allocating the arrays `m.i`, `m.j`, `m.a` and for filling their values.
6. The `mr` function object is applied at lines 15–37. Details are described further in the text.
7. The variable `lambda` for storing the resulting eigenvalue is defined at line 39.
8. At line 40, the `pm` function object is defined. It is responsible for computing the eigenvalue and deallocating the arrays.
9. The `pm` function object is applied at lines 42–52. Details are described further in the text.
10. The computed eigenvalue is printed out at line 54.

Since we wanted to evaluate the `for_id` algorithm, we created multiple instances (cases) of the program that are listed in Table 1. In the cases C^f_{16}–C^d_{32}, the function call operators are invoked directly at lines 15–22 and 42–49, which corresponds to the classical approach where data types are resolved at compile time. However, in the case C^*_*, the `for_id` algorithm was utilized (lines 34 and 51), where:

1. the `fp_types` and `ind_types` type sequences are defined at lines 2 and 3,
2. the floating-point type *id* is selected by the third command line option at line 24,
3. the indexing type *id* is selected according to the number of matrix rows/columns at lines[5] 26–32.

Moreover, we distinguish two sub-cases of C^*_*—C^f_* and C^d_* for *single* and *double* precision computation selected at run time, respectively.

Finally, we used the following definitions of the `MatrixReader` and `PowerMethod` classes:

```
class MatrixReader {
  public:
    MatrixReader(Matrix& m, std::ifstream& ifs) : m_(m), ifs_(ifs) { }

    template <typename F, typename I>
    void operator()() {
      I* i = new I[m_.z];
```

[5]The row and column indexes are integer numbers between 0 and $\texttt{m.n} - 1$, thus, we need an unsigned integer data type of width $b(\texttt{m.n})$ bits (1).

```cpp
         I* j = new I[m_.z];
         F* a = new F[m_.z];

         for (uint64_t k = 0; k < m_.z; ++k) {
            ifs_ >> i[k] >> j[k] >> a[k];
            i[k]--; // 1- to 0-based indexing shift
            j[k]--;
         }

         m_.i = static_cast<void*>(i);
         m_.j = static_cast<void*>(j);
         m_.a = static_cast<void*>(a);
      }

   private:
      Matrix& m_;
      std::ifstream& ifs_;
};

class PowerMethod {
   public:
      PowerMethod(const Matrix& m, uint64_t q, double& lambda)
         : m_(m), q_(q), lambda_(lambda) { }

      template <typename F, typename I>
      void operator()() {
         I* i = static_cast<I*>(m_.i);
         I* j = static_cast<I*>(m_.j);
         F* a = static_cast<F*>(m_.a);

         std::vector<F> x(m_.n, 1.0), y(m_.n);
         F lambda = 0.0;

         // power method iterations:
         do {
            std::fill(y.begin(), y.end(), 0.0);
            for (uint64_t k = 0; k < m_.z; ++k) {
               y[i[k]] += a[k] * x[j[k]];
               if (i[k] != j[k])
                  y[j[k]] += a[k] * x[i[k]];
            }
            lambda = 0.0;
            for (uint64_t k = 0; k < m_.n; ++k)
               if (fabs(y[k]) > fabs(lambda))
                  lambda = y[k];
            for (uint64_t k = 0; k < m_.n; ++k)
               y[k] /= lambda;
            std::copy(y.begin(), y.end(), x.begin());
         } while (--q_ > 0);

         lambda_ = static_cast<double>(lambda);

         delete[] i;
         delete[] j;
         delete[] a;
   }

   private:
      const Matrix& m_;
      uint64_t q_;
      double& lambda_;
};
```

Note that:

Table 2: Compatibility of the `for_id`-based case C^*_* of the test program with different compilers.

Compiler vendor	Compiler version	Boost version	Processor architecture	Operating system	Compilation errors
Cray	8.0.7	1.47.0	AMD x86_64	Linux	YES
GNU	4.7.1	1.47.0	AMD x86_64	Linux	NO
IBM	11.1	1.40.0	IBM POWER7	AIX	YES
Intel	12.1.5	1.47.0	AMD x86_64	Linux	NO
Microsoft	16.0.0.30319 01	1.48.0	Intel x86_64	Windows	YES
PathScale	4.0.12.1	1.47.0	AMD x86_64	Linux	NO
PGI	12.5-0	1.47.0	AMD x86_64	Linux	NO

Table 3: Compilation time of the test program in seconds; average results of 10 measurements.

Action	C^f_{16}	C^*_*
preprocessing, compilation, linking	0.93	1.17
compilation only	0.76	0.94

1. We used the power method for computing the dominant eigenvalue (lines 42–56).
2. We used `void` pointers for storing data whose types are not known until run time (lines 17–19).
3. We used *pass-by-value* and *pass-by-reference* constructor arguments to pass data *to* and *from* the function objects, respectively (lines 3 and 29).

4.2 Results and Discussion

We compiled the `for_id`-based case C^*_* of the test program with different compilers. The results are presented in Table 2. The Intel compiler required the `-std=c++0x` command line option. The Cray compiler threw multiple errors, which involved rvalue references and the `BOOST_STATIC_ASSERT` macro. The IBM and Microsoft compilers were not able to handle the `f.template operator()<T>()` construct and hence they were not compliant with the C++ Standard at this point[6] (see [ISO03, §13.5/4 and §14.2/4] for more details).

For the measurements described below, we used the GNU C++ compiler version 4.4.4. We first compared the compilation time—the results are presented in Table 3. When the program was built completely, the compilation time of C^*_* was 25 percent higher compared to C^f_{16}. When the program was compiled only, the increase was 31 percent.

Next, we compared the sizes of output files—the results are presented in Table 4. The executable file size of C^*_* is 83 percent higher compared to C^f_{16}.

For comparison of the memory requirements of the program instances, we used 3 real symmetric matrices from the University of Florida Sparse Matrix Collection [DH11]; their names and characteristics are contained in Table 5. We measured the memory size of the matrix and vector data structures and compared them separately for single and double precision computations—the results are shown in Fig. 1. It is clear that the program instances based on `for_id` always require the minimum amount of memory, because an optimal data type is used for indexes (if we included program instances using the `uint64_t` data type into our measurements, this advantage would be even more significant).

Lastly, we measured the computational overhead of the `for_id` algorithm. We used the `clock_gettime` POSIX function to get the actual time values in nanoseconds at 3 places:

[6]The workaround here might be to use a fixed-name member function instead of the function call operator. We have not tested this possibility.

Table 4: Sizes of the compiled files in kilobytes.

File	C^f_{16}	C^*_*
executable	53.2	97.6
object	104.7	211.1

Table 5: Characteristics of matrices used for experiments.

Number of	nos1	thread	ldoor
rows	237	$29.7 \cdot 10^3$	$952.2 \cdot 10^3$
nonzero elements	627	$2.3 \cdot 10^6$	$23.7 \cdot 10^6$

1. at line 41 in the `main()` function,
2. at the very beginning of the function call operator of the `PowerMethod` class (line 33)
3. at line 53 in the `main()` function.

The difference of the first and the second time values is equal to the time overhead of the *invocation* of the `pm`'s function call operator. The difference of the first and the third time values is equal to the duration of the *application* of the `pm`'s function call operator, that is, the whole run of the power method. The statistical information for the performed measurements are summarized in Table 6. It is clear that the time overhead introduced by the `for_id` algorithm is relatively high—the invocation of the function call operator takes 2.5 times longer than in the cases where this operator is called directly. However, in the context of the whole program run, this overhead is insignificant, since it is of five orders of magnitude smaller than the duration of a single power method iteration.

4.3 Compilation Scalability

Besides the experiments described heretofore, we also measured the compilation scalability of `for_id`. The term *compilation scalability* denotes the response of a C++ compiler to the growing number of problem dimensions as well as the increasing length of type sequences. Particularly, we focused on the time and memory requirements of the compilation process. In order to carry out measurements, we created special source code:

1. We defined the following type sequences, each of the same length L:

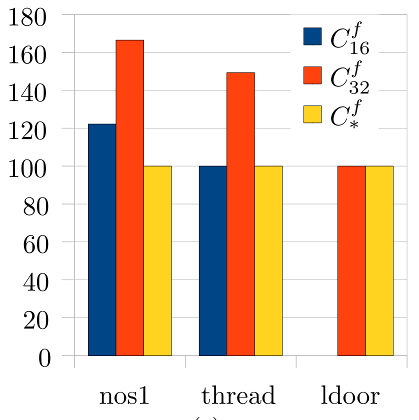
$$\mathcal{T}_i = \{t_i^1, \dots, t_i^L\} \quad \text{for} \quad i = 1, \dots, d.$$

2. We defined the function object `f` as follows:

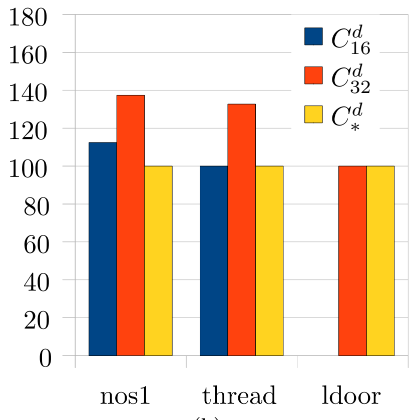
```
struct {
    template <typename t1, ... , typename td>
    void operator()() {
        ... // some code
    }
} f;
```

3. We invoked the `for_id` function:

```
int id1, ... , idd; // values do not matter
for_id<T1, ... ,Td>(f, id1, ... , idd);
```



(a)



(b)

Figure 1: Comparison of program's memory requirements of the matrix and vector data structures in percents for different matrices and for computations in *single* (a) and *double* (b) floating-point precision.

Table 6: Time differences for the `pm`'s function call operator in nanoseconds. Statistical information was gathered from 200 measurements with the thread matrix. The number of iterations of the power method was set to 10.

Action	Statistic	C^f_{16}	C^d_{32}	C^f_*
invocation	mean value	191.1	191.0	477.9
	median	186.0	187.0	470.0
	standard deviation	23.0	25.1	43.1
application	mean value	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$
	median	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$
	standard deviation	$6.9 \cdot 10^6$	$7.7 \cdot 10^6$	$6.6 \cdot 10^6$

(Recall that we were interested in the compilation process only. Resulting programs were not executed at run time and therefore the code inside the function call operator and the values of particular type *id*s were irrelevant here.)

We performed the experiments for both d and L ranging from 1 to 6. The number of instances of the function call operator that needed to be created by the compiler within this domain is shown in Table 7 (it is equal to L^d).

We used the GNU C++ compiler version 4.4.6 for all experiments described in this section. To measure compilation times we utilized the `time` command available in the Linux operating system. The results of these measurements are presented in Table 8, wherein N/A means that we were not able to obtain the corresponding value due to time/memory restrictions.

One can observe that the compilation time grows rapidly with increasing values of both parameters d and L. Moreover, the compilation time grows considerably faster than the number of instances. To show this effect, we present—in Table 9—*relative compilation time per instance*, which equals the ratio of the overall compilation time from Table 8 and the number of instances from Table 7. There exists a compilation overhead that is not related to `for_id` and hence the values of relative compilation time are not relevant for small d and L. Therefore, we restrict the results presented in Table 9 to those corresponding to the overall compilation time longer than 1 second.

Finally, we also measured the memory requirements of the compiler. We used the Valgrind tool—particularly its heap profiler Massif—for this experiment [NS07]. The results are presented in Table 10. For $d = L = 6$ the memory requirements exceeded 4 GB.

Table 7: The number of instances of the function call operator that were created by the compiler for a range of problem dimensions and length of type sequences.

	Problem dimension d					
Length L	1	2	3	4	5	6
1	1	1	1	1	1	1
2	2	4	8	16	32	64
3	3	9	27	81	243	729
4	4	16	64	256	1024	4096
5	5	25	125	625	3125	15625
6	6	36	216	1296	7776	46656

Table 8: Compilation times in seconds for a range of problem dimensions and length of type sequences.

	Problem dimension d					
Length L	1	2	3	4	5	6
1	0.4	0.4	0.4	0.4	0.4	0.4
2	0.4	0.4	0.5	0.5	0.6	0.8
3	0.4	0.4	0.5	0.9	1.7	5.6
4	0.4	0.5	0.7	1.7	8.1	109.6
5	0.4	0.5	1.0	3.9	57.3	2405.2
6	0.4	0.6	1.4	11.0	514.5	N/A

5 Conclusions

The contribution of this paper is a new method that allows users to select data types for a piece of templated C++ code at run time with the minimal sustained complexity of code branching. The only requirement for such a piece of code is that it has to be in a form of a templated fuction call operator of some function object. The following conclusions can be drawn from the results of the performed experiments:

- The use of `for_id` allows users to select the floating-point precision for computations at run time without the need of program recompilation.
- The use of `for_id` allows the best utilization of the computer memory for data structures that contain indexes.
- The use of `for_id` requires higher computational and memory resources for the compilation process, especially for longer type sequences and higher problem dimensions.
- The use of `for_id` results in a bigger executable file, that is, in a bigger program's code segment.
- The use of `for_id` imposes a run-time overhead on the application of the function object.

The drawbacks seemingly prevail over the advantages. However, we need to realize that in typical real-world situations these drawbacks will be insignificant, since:

- Programs are usually compiled only once and then executed multiple times, and/or their compilation time is usually much smaller than their execution time.
- The size of the code segments of running program instances are usually much smaller than the size of their data segments.

Table 9: Relative compilation times per instance in milliseconds for a range of problem dimensions and length of type sequences. The values are presented only for overall compilation times longer than 1 second.

	Problem dimension d					
Length L	1	2	3	4	5	6
3					7.1	7.6
4				6.5	7.9	26.8
5			7.6	6.2	18.3	153.9
6			6.5	8.5	66.2	N/A

Table 10: Memory requirements of a compiler in megabytes for a range of problem dimensions and length of type sequences.

	Problem dimension d					
Length L	1	2	3	4	5	6
1	157.0	158.0	159.2	159.0	160.2	161.3
2	158.0	160.3	162.3	166.7	173.1	185.0
3	159.1	162.3	168.9	184.0	226.5	351.1
4	160.2	165.6	179.7	225.5	401.1	635.4
5	161.3	170.0	194.8	304.6	524.9	1842.8
6	162.3	174.3	215.6	407.8	1094.0	N/A

- The execution time of the templated code is usually of several orders of magnitude longer than the run-time overhead of its invocation.

The purpose of our rather artificial test program was to evaluate the `for_id` algorithm. However, we have also successfully integrated `for_id` into an existing HPC code, namely the code that solves *symmetry-adapted no-core shell model* problems [DSB+07b, DSB+07a, DSD+08]. These problems are extremely memory-demanding and the limit for the size of the problem that can be solved on a particular HPC system is given rather by the amount of available memory than by the computational power of its processors. Inside the code, we have utilized `for_id` for many different tasks, including a sparse matrix-vector multiplication or a parallel file input/output of sparse matrices.

The use of `for_id` allows to eliminate wasting of data memory for applications that use many different data structures containing arrays of indexes. In addition, it also allows to compile such applications only once even if the types of indexes of submitted data and/or the floating-point precision of computations vary for various runs. This may be especially useful for HPC programs that run on massively parallel supercomputers. Another example where `for_id` might be useful as well is the implementation of generic image algorithms as used inside GIL (see Section 2).

Acknowledgements

This work was supported by the Czech Science Foundation under Grant No. P202/12/2011, by the U.S. National Science Foundation under Grant No. OCI-0904874, and by the U.S. Department of Energy under Grant No. DOE-0904874.

A Header Files

The header files required for the `for_id`, `for_id_impl`, and `execute` definitions in Section 3:

```
#include <stdexcept>
```

```
#include <boost/mpl/begin_end.hpp>
#include <boost/mpl/deref.hpp>
#include <boost/mpl/distance.hpp>
#include <boost/mpl/find.hpp>
#include <boost/mpl/next_prior.hpp>
#include <boost/mpl/vector.hpp>
```

We further suppose that these definitions are placed in the separate header file `for_id.h` and that this header file is in the same directory as the test program. The header files required for the compilation of the test program defined in Section 4 are the following:

```
#include <algorithm>
#include <cstring>
#include <fstream>
#include <iostream>
#include <vector>

#include <stdint.h>

#include <boost/lexical_cast.hpp>

#include "for_id.h"
```

References

[AG04] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 2001.

[BBB$^+$10] Satish Balay, Jed Brown, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.2, Argonne National Laboratory, 2010.

[BJ] L. Bourdev and H. Jin. Generic Image Library. `http://opensource.adobe.com/gil` (accessed December, 2011).

[BJ11] Lubomir Bourdev and Jaakko Järvi. Efficient run-time dispatching in generic programming with minimal code bloat. *Science of Computer Programming*, 76(4):243–257, 2011. `doi:10.1016/j.scico.2008.06.003`.

[BN94] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1994.

[BPR96] Ronald F. Boisvert, Roldan Pozo, and Karin Remington. The Matrix Market Exchange Formats: Initial Design. Technical Report NISTIR 5935, National Institute of Standards and Technology, Dec. 1996.

[DH11] T. A. Davis and Y. F. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1), 2011. `doi:10.1145/2049662.2049663`.

[DSB$^+$07a] T. Dytrych, K. D. Sviratcheva, C. Bahri, J. P. Draayer, and J. P. Vary. Dominant role of symplectic symmetry in ab initio no-core shell model results for light nuclei. *Physical Review C*, 76(1):014315, 2007. `doi:10.1103/PhysRevC.76.014315`.

[DSB+07b] T. Dytrych, K. D. Sviratcheva, C. Bahri, J. P. Draayer, and J. P. Vary. Evidence for symplectic symmetry in ab initio no-core shell model results for light nuclei. *Physical Review Letters*, 98:162503, 2007. `doi:10.1103/PhysRevLett.98.162503`.

[DSD+08] T. Dytrych, K. D. Sviratcheva, J. P. Draayer, C. Bahri, and J. P. Vary. Ab initio symplectic no-core shell model. *Journal of Physics G: Nuclear and Particle Physics*, 35(12):123101, 2008. `doi:10.1088/0954-3899/35/12/123101`.

[Gen11] Davide Di Gennaro. *Advanced C++ Metaprogramming.* CreateSpace, 2011.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen, version 3.0.1, 2010. `http://eigen.tuxfamily.org` (accessed July, 2011).

[HW03] Michael A. Heroux and James M. Willenbring. Trilinos users guide. Technical Report SAND2003-2952, Sandia National Laboratories, 2003.

[ISO03] *ISO/IEC 14882:2003: Programming languages: C++.* 2003.

[LTDD12] Daniel Langr, Pavel Tvrdík, Tomáš Dytrych, and Jerry P. Draayer. Fake Run-Time Selection of Template Arguments in C++. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns (50th International Conference, TOOLS 2012)*, volume 7304 of *Lecture Notes in Computer Science*, pages 140–154. Springer Berlin Heidelberg, 2012. `doi:10.1007/978-3-642-30561-0_11`.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. URL: `http://dx.doi.org/10.1145/1250734.1250746`, `doi:10.1145/1250734.1250746`.

[Ope11] *OpenFOAM User Guide, Version 2.0.0.* 2011.

[Pra01] Stephen Prata. *C++ Primer Plus (Fourth Edition).* Sams, Indianapolis, IN, USA, 4th edition, 2001.

[Str00] Bjarne Stroustrup. *The C++ Programming Language: Special Edition.* Addison-Wesley Professional, 3 edition, February 2000. URL: `http://www.worldcat.org/isbn/0201700735`.

[Thea] The Boost MPL Library. `http://www.boost.org/doc/libs/1_48_0/libs/mpl/doc/index.html` (accessed December 12, 2012).

[Theb] The HDF Group. Hierarchical data format version 5, 2000-2013. `http://www.hdfgroup.org/HDF5/` (accessed June 3, 2013).

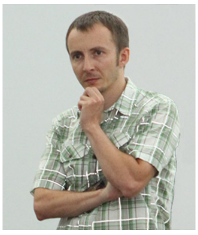
[VDY05] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521–530, 2005. `doi:10.1088/1742-6596/16/1/071`.

[VJ02] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates—The Complete Guide.* Addison-Wesley, 2002.

Authors' Biographies

Daniel Langr is a Ph.D. student and a researcher at the Czech Technical University in Prague, Czech Republic, where he is a member of the Parallel Computing Group at the Faculty of Information Technologies. His main research interests include parallel I/O and visualization algorithms for sparse matrices, memory-efficient storage formats for sparse matrices, and generic programming and template metaprogramming in C++. Besides others, he focuses on discovering new approaches to high-performance-computing algorithms based on modern object-oriented technologies. Contact him at langrd@fit.cvut.cz.

Pavel Tvrdík received his M.Eng. and Ph.D. in Computer Science from the Czech Technical University in Prague, in 1980 and 1991, respectively. Currently, he is a full professor at the Department of Computer Systems, Faculty of Information Technology, Czech Technical University in Prague. His research interests include parallel computer architectures and algorithms and cluster computing. Contact him at pavel.tvrdik@fit.cvut.cz.



Tomáš Dytrych is a senior research associate at the Department of Physics and Astronomy, Louisiana State University, USA. His research interests include physics of atomic nuclei, computational group theory, general-purpose computing on graphics processing units, and application of modern programming techniques based on template capabilities of C++ for high performance computing of nuclear structure. Contact him at tdytrych@phys.lsu.edu.

Jerry P. Draayer is the president and CEO of the Southwestern Universities Research Association (SURA) and the Roy P. Daniels Professor of Physics at the Department of Physics and Astronomy, Louisiana State University, USA. His research interest include the low-energy structure of atomic nuclei. Another area of interest is nonlinear phenomena, in particular, analytic and numeric tools for dealing with nonlinear processes, consequences of nonlinear couplings in physical systems, and how nonlinear effects are manifest in atomic nuclei. Contact him at draayer@phys.lsu.edu.