DEVELOPMENT IN WAVE FUNCTION METHODS MADE EASY

WITH IRPF90 AND THE QUANTUM PACKAGE

Anthony Scemama, & E. Giner, M. Caffarel, T. Applencourt, Y. Garniron
23/03/2016

Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse
Scientific codes need **speed** → Fortran / C

- Low-level languages: difficult to maintain
- High-level features of modern Fortran (**matmul**, array syntax, derived types, ...) or C++ (objects, STL) can kill the efficiency

We need to hide the code complexity and keep the code efficient.
A simple solution: use multiple languages.

- High-level: text parsing, global code architecture, ...
- Low-level: computation
- Meta-programming: generate low-level code with a higher-level language

Problem addressed here

Make code in the low-level language easy to write and maintain
Programming with Implicit Reference to Parameters (IRP)

Motivations

The IRP method

The IRPF90 code generator

Quantum Package
PROGRAMMING WITH IMPLICIT REFERENCE TO PARAMETERS (IRP)
Programming with Implicit Reference to Parameters (IRP)

Motivations

The IRP method

The IRPF90 code generator
A program is a function of its input data:

\[ \text{output} = \text{program}(\text{input}) \]

A program can be represented as a production tree where

- The root is the output
- The leaves are the input data
- The nodes are the intermediate variables
- The edges represent the relation needs/needed by
Example: Production tree of $t(u(d_1, d_2), v(u(d_3, d_4), w(d_5)))$

$u(x, y) = x + y + 1$

$v(x, y) = x + y + 2$

$w(x) = x + 3$

$t(x, y) = x + y + 4$
program compute_t
  implicit none
  integer :: d1, d2, d3, d4, d5
  integer :: u, v, w, t

  call read_data(d1,d2,d3,d4,d5) ! t
    ! / \
  call compute_u(d3,d4,u) ! u v
  call compute_w(d5,w) ! / / / \ 
  call compute_v(u,w,v) ! d1 d2 u w
  call compute_u(d1,d2,u) ! / / \ \ 
  call compute_t(u,v,t) ! d3 d4 d5

  write(*,*) , "t=" , t
end program
Imperative programming (wikipedia)

[...] programming paradigm that uses statements that change a program’s state.

- The code expresses the exploration of the production tree
- The routines have to be called in the correct order
- The values of variables are time-dependent
Sources of complexity

1. Time-dependence of the data (mutable data)
2. Handling the complexity of the production tree
Functional programming (wikipedia)

[...] programming paradigm [...] that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

No time-dependence (immutable data) $\implies$ reduced complexity
program compute_t
  implicit none
  integer :: d1, d2, d3, d4, d5
  integer :: u, v, w, t

  call read_data(d1, d2, d3, d4, d5)

  write(*,*) , "t=", t( u(d1, d2), v( u(d3, d4), w(d5) ) )
end program

• Instead of telling *what to do*, we express *what we want*
• The programmer doesn’t handle the execution sequence

No time-dependence left
2. COMPLEXITY OF THE PRODUCTION TREE

Production tree of $\Psi$ in QMC=Chem: 149 nodes / 689 edges
2. Complexity of the production tree

1. The programmers need to have the *global knowledge* of the production tree: Production trees are usually too complex to be handled by humans.
2. Programmers may not be sure that their modification did not break some other part.
3. Collaborative work is difficult: any user can alter the production tree.
Express the needed entities for each node:

- \( t \rightarrow u_1 \) and \( v \)
- \( u_1 \rightarrow d_1 \) and \( d_2 \)
- \( v \rightarrow u_2 \) and \( w \)
- \( u_2 \rightarrow d_3 \) and \( d_4 \)
- \( w \rightarrow d_5 \)

The information is now *local* and easy to handle.
program compute_t
    integer, external :: t
    write(*,*), "t=" , t()
end program

integer function t()
    integer, external :: u1, v
    t = u1() + v() + 4
end

integer function v()
    integer, external :: u2, w
    v = u2() + w() + 2
end

integer function w()
    integer :: d1,d2,d3,d4,d5
    call read_data(d1,d2,d3,d4,d5)
    w = d5 + 3
end

integer function f_u(x,y)
    integer, intent(in) :: x,y
    f_u = x+y+1
end

integer function u1()
    integer :: d1,d2,d3,d4,d5
    integer, external :: f_u
    call read_data(d1,d2,d3,d4,d5)
    u1 = f_u(d1,d2)
end

integer function u2()
    integer :: d1,d2,d3,d4,d5
    integer, external :: f_u
    call read_data(d1,d2,d3,d4,d5)
    u2 = f_u(d3,d4)
end
• The global production tree is not known by the programmer
• The program is easy to write
• Any change of dependencies will be handled properly automatically

But: The same data will be recomputed multiple times.
Simple solution: Lazy evaluation using memo functions.
Programming with Implicit Reference to Parameters (IRP)

Motivations

The IRP method

The IRPF90 code generator
**Entity**  Node of the production tree

**Builder**  Subroutine that builds a valid value of an entity from its dependencies

**Valid**  Fully initialized with meaningful values

**Provider**  Subroutine with no argument which guarantees to return a valid value of an entity

**Rules of IRP**

1. Each entity has only one provider
2. Before using an entity, its provider has to be called

---

François Colonna : "IRP programming : an efficient way to reduce inter-module coupling ", DOI: 10.13140/RG.2.1.3833.0406
program test
  use entities
  implicit none
  call provide_t
  print *, "t=" , t
end program

module entities
  ! Entities
  integer :: u1, u2, v, w, t
  logical :: u1_is_built = .False.
  logical :: u2_is_built = .False.
  logical :: v_is_built = .False.
  logical :: w_is_built = .False.
  logical :: t_is_built = .False.

  ! Leaves
  integer :: d1, d2, d3, d4, d5
  logical :: d_is_built = .False.
end module

subroutine provide_t
  use entities
  implicit none
  if (.not. t_is_built) then
    call provide_u1
    call provide_v
    call build_t(u1,v,t)
    t_is_built = .True.
  endif
end subroutine provide_t

subroutine build_t(x,y,result)
  implicit none
  integer, intent(in) :: x, y
  integer, intent(out) :: result
  result = x + y + 4
end subroutine build_t
SUMMARY

With the IRP method:

1. Code is easy to develop for a new developer: Adding a new feature only requires to know the names of the needed entities
2. If one developer changes the dependence tree, the others will not be affected: collaborative work is simple
3. Forces to write clear code: one builder builds only one thing
4. Forces to write efficient code (spatial and temporal localities are good)

But in real life:

1. A lot more typing is required
2. Programmers are lazy
Programming with Implicit Reference to Parameters (IRP)

Motivations

The IRP method

The IRPF90 code generator
• Extends Fortran with additional keywords
• Fortran code generator (source-to-source compiler)
• Writes all the mechanical IRP code

Useful features:
• Automatic Makefile generation
• Automatic Documentation
• Text editor integration
• Some Introspection
• Meta programming
• Some features targeted for HPC

http://irpf90.ups-tlse.fr
https://github.com/scemama/irpf90
program irp_example
    print *, 't=', t
end

BEGIN_PROVIDER [ integer, t ]
    t = u1+v+4
END_PROVIDER

BEGIN_PROVIDER [ integer, w ]
    w = d5+3
END_PROVIDER

BEGIN_PROVIDER [ integer, v ]
    v = u2+w+2
END_PROVIDER

integer function fu(x,y)
    integer, intent(in) :: x,y
    fu = x+y+1
end function
BEGIN_PROVIDER [ double precision, A, (dim1, 3) ]
...
END_PROVIDER

• Allocation of IRP arrays done automatically
• Dimensioning variables can be IRP entities, provided before the memory allocation
• FREE keyword to force to free memory. Invalidates the entity.
BEGIN_PROVIDER [ double precision, Fock_matrix_beta_mo, &
    (mo_tot_num_align,mo_tot_num_num) ]

implicit none
BEGIN_DOC
! Fock matrix on the MO basis
END_DOC
...
END_PROVIDER

$ irpman fock_matrix_beta_mo
IRPF90 entities(1)  fock_matrix_beta_mo  IRPF90 entities(1)

Declaration

double precision, allocatable :: fock_matrix_beta_mo (mo_tot_num_align,mo_tot_num)

Description

Fock matrix on the MO basis

File

Fock_matrix.irp.f

Needs

ao_num
fock_matrix_alpha_ao
mo_coef
mo_tot_num
mo_tot_num_align

Needed by

fock_matrix_mo

IRPF90 entities(1)  fock_matrix_beta_mo  IRPF90 entities(1)
MOVIE
Iterative processes involve cyclic dependencies

TOUCH A: A is valid, but everything that needs A is invalidated.
(a) Everything is valid
(b) $x$ is modified
(c) $x$ TOUCHed
Many other features

- Assert keyword, Templates
- Variables can be declared anywhere
- $+=, -=, *= $ operators
- Dependencies are known by IRPF90 → Makefiles are built automatically
- Array alignment, Variable substitution
- Codelet generation
- TSC Profiler
- Thread safety (OpenMP)
- Syntax highlighting in Vim
- Generation of tags to navigate in the code
- No problem using external libraries (MKL, MPI, etc)
- ...
QUANTUM PACKAGE
IRPF90 library for **post-HF** quantum chemistry

- Open Source (GPL): contributors are welcome!
- Hosted on GitHub: [https://github.com/LCPQ/quantum_package](https://github.com/LCPQ/quantum_package)
- Goal: easy for the programmer
- Long term objective: Massively parallel post-HF
WHY ANOTHER QUANTUM PACKAGE?

- Full-CI: $\mathcal{O}(N!)$ (formally)
- CAS-CI: Full-CI in a very small space
- Complete: easy (integral $\leftrightarrow$ determinant) mapping
- Integral-driven algorithms: very efficient

**Perturbative selection (old idea re-discovered regularly)**

1. Don’t explore the complete CI space, but select determinants on-the-fly (CIPSI) with perturbation theory.
2. Use PT2 to estimate the missing part
Consequences:

- Much larger spaces can be explored
- **Selected**: difficult (integral $\leftrightarrow$ determinant) mapping
- **Determinant-driven algorithms**: less efficient

**But**

The drastic reduction of the space makes the selected approach more efficient
ALGORITHM

1. Start with $|\psi_0\rangle = |\text{HF}\rangle$
2. $\forall \{ |i\rangle \} \notin \psi_n \text{ but } |i\rangle \in \{ T_{\text{SD}} |\psi_n\rangle \}$, compute $e_i = \frac{\langle i | \mathcal{H} |\psi_n\rangle^2}{E(\psi_n) - \langle i | \mathcal{H} | i \rangle}$
3. if $|e_i| > \epsilon_n$, select $|i\rangle$
4. Estimated energy: $E(\psi_n) + E(PT2)_n = E(\psi_n) + \sum_i e_i$
5. $\psi_{n+1} = \psi_n + \sum_{i \text{ (selected)}} c_i |i\rangle$
6. Minimize $E(\psi_{n+1})$ (Davidson)
7. Choose $\epsilon_{n+1} < \epsilon_n$
8. Go to step 2
When $n \to \infty$, $E(PT2)_{n=\infty} = 0$, so the complete CI problem is solved.

Every CI problem can be solved by iterative perturbative selection.

Perturbatively Selected CI is not a method, but an algorithm. It can be applied to:

- Full-CI
- CISD, CISDT, CISTDQ, ...
- CAS-CI, MR-CI
- ...

Implies determinant-driven algorithms $\implies$ Requires an Efficient implementation of Slater Rules
Post-Full-CI: QMC/Full-CI (E. Giner, T. Applencourt, M. Caffarel)

![Graph showing energy (a.u.) as a function of 1/n for different basis sets. The graph includes points for Full-CI, DMC/CIPSI, and the estimated exact. The x-axis represents 1/n with values from 0.0 to 0.5, and the y-axis represents energy (a.u.) with values ranging from -76.44 to -76.39. The basis sets are labeled as DZ, 6Z, 5Z, QZ, TZ, and cc-pCVnZ, with n ranging from 5 to 6.]
**MR-CCSD**: (E. Giner, G. David, J.-P. Malrieu)

TABLE VII. Symmetric dissociation of the water molecule, cc-pVDZ basis set. The FCI total energy\(^5\) is given in \(E_h\), and the deviations to this reference are given in \(mE_h\). Comparison with Mukherjee’s state specific MR-CC values \((E_{\text{Mk-MR-CCSD}} - E_{\text{FCI}})\) obtained from Ref. 40.

<table>
<thead>
<tr>
<th>(R (R_e))</th>
<th>(E_{\text{CAS-CISD}} - E_{\text{FCI}})</th>
<th>(E_{\text{Mk-MR-CCSD}} - E_{\text{FCI}})</th>
<th>(E_{\text{MR-CCSD}} - E_{\text{FCI}})</th>
<th>FCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (R_e)</td>
<td>4.923</td>
<td>2.909</td>
<td>1.407</td>
<td>(-76.241,860)</td>
</tr>
<tr>
<td>1.5 (R_e)</td>
<td>4.674</td>
<td>4.817</td>
<td>1.248</td>
<td>(-76.072,348)</td>
</tr>
<tr>
<td>2.0 (R_e)</td>
<td>3.665</td>
<td>6.485</td>
<td>0.855</td>
<td>(-75.951,665)</td>
</tr>
<tr>
<td>2.5 (R_e)</td>
<td>3.097</td>
<td>5.672</td>
<td>0.763</td>
<td>(-75.917,991)</td>
</tr>
<tr>
<td>3.0 (R_e)</td>
<td>2.959</td>
<td>3.987</td>
<td>0.845</td>
<td>(-75.911,946)</td>
</tr>
<tr>
<td>NPE</td>
<td>1.964</td>
<td>3.576</td>
<td>0.644</td>
<td></td>
</tr>
</tbody>
</table>
(Work in progress)
• $C_2$ Full-CI demo
• Simple Hartree-Fock