IRPF90 : a Fortran code generator for HPC

Anthony Scemama¹ <scemama@irsamc.ups-tlse.fr>
François Colonna²

¹ Laboratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)
² Laboratoire de Chimie Théorique (Paris)
Scientific codes

- Scientific codes need *speed* -> Fortran
- Fortran is a low level language -> difficult to maintain
- High-level features of Fortran 95 kill the efficiency (pointers, array syntax, etc) -> not a good solution

- Less effort in program development and good efficiency with Python/Numpy or Python/F2Py
- Other option : use Python to write low level Fortran code
What is a scientific code?

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 nodes are the variables
- The vertices represent the relation *needs/needed by*

Example:

\[
\begin{align*}
  u(x, y) &= x + y + 1 \\
  v(x, y) &= x + y + 2 \\
  w(x) &= x + 3 \\
  t(x, y) &= x + y + 4
\end{align*}
\]

What is the production tree of \( t(u(d1,d2), v(u(d3,d4), w(d5))) \)?
What is the production tree of \( t(\ u(d1,d2),\ v(\ u(d3,d4),\ w(d5))\ )\)?

\[
\begin{align*}
    u(x,y) &= x + y + 1 \\
    v(x,y) &= x + y + 2 \\
    w(x) &= x + 3 \\
    t(x,y) &= x + y + 4
\end{align*}
\]
def fu(x, y): return x+y+1
def fv(x, y): return x+y+2
def fw(x) : return x+3
def ft(x, y): return x+y+4

def input_data():
    # ...
    return d1, d2, d3, d4, d5

def main():
    # t
    d1,d2,d3,d4,d5 = input_data()# / \ # u1 v
    u1 = fu(d1, d2) # / | | \ # d1 d2 u2 w
    u2 = fu(d3, d4) # / | | \ # d3 d4 d5
    w = fw(d5)
    v = fv(u2, w)
    print ft(u1, v)
Difficulties

The subroutines need to be called in the correct order:

- The programmers need have the knowledge of the production tree
- Production trees are usually too complex to be handled by humans
- Collaborative work is difficult: any user can alter the production tree
- Programmers may not be sure that their modification did not break some other part
IRP way of programming

- There is only one way to build a value: by calling its provider
- Provider:
  - If the value is already built: return the previous value (memo function)
  - Otherwise: call the providers of the needed entities and then build the value

Advantages:
- The provider guarantees that the value is valid
- Only a local knowledge of the production tree: the needed entities
- This is equivalent to calling $t(u(d1,d2), v(u(d3,d4), w(d5)))$. As there is only one set of possible parameters for each function (the needed entities are always the same), the parameters can be embedded inside the functions.

IRP: functions with Implicit Reference to Parameters
Example

```python
import sys

def irp(f):
    """All the IRP entities are represented inside a common class.
    This function generates the provider of an entity.
    f is the function that builds the entity.
    """

    # Switch the current environment to the class containing the
    # IRP entities
    locals = sys._getframe(1).f_locals

    # Name of the memo value : _f
    cache = "_"+f.__name__

    # Generic provider
```
def provider(self):
    # Check if self._f exists
    try:
        result = getattr(self, cache)
        print " -- Already built " + f.__name__
    except AttributeError:
        # If self._f doesn't exist, build it
        print " -> Building " + f.__name__
        result = f(self)
        setattr(self, cache, result)
        print " <- Done building " + f.__name__
    # Return self._f
    return result

# Return a class property to call the provider
return property(fget=provider)
class MyProgram(object):

    def u1(self):  # u(x,y) = x + y + 1
        return self.d1 + self.d2 + 1
    u1 = irp(u1)

    def u2(self):  # u(x,y) = x + y + 1
        return self.d3 + self.d4 + 1
    u2 = irp(u2)

    def v(self):  # v(x,y) = x + y + 2
        return self.u2 + self.w + 2
    v = irp(v)

    def w(self):  # w(x) = x + 3
        return self.d5 + 3
    w = irp(w)
def t(self):  # t(x,y) = x + y + 4
    return self.u1 + self.v + 4

    t = irp(t)

def d(self):
    return range(1,6)

d  = irp(d)

d1 = property(lambda self: self.d[0])
d2 = property(lambda self: self.d[1])
d3 = property(lambda self: self.d[2])
d4 = property(lambda self: self.d[3])
d5 = property(lambda self: self.d[4])

def main():
    p = MyProgram()
    print "u1 : " ; print p.u1
    print "t  : " ; print p.t
main()

u1 :
  --> Building u1
  --> Building d
  <-- Done building d
  -- Already built d
  <-- Done building u1

4

v

#        t
#       /    \
#     u1      v
#   /  |      |  \
# d1   d2    u2   w
#          /   \
#        d3    d4    d5

w

3

4

 Already built d
<- Done building w
<- Done building v
<- Done building t
26
How to do this with Fortran?

Fortran doesn't have exceptions, introspection, properties, etc. Solution IRPF90:

- Add a few keywords to Fortran
- Use Python to read the code
- Python builds the dependence tree
- Python writes the missing Fortran source lines to handle the tree

Example with IRPF90

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

BEGIN_PROVIDER [ integer, w ]
  w = d5+3
```
BEGIN_PROVIDER [ integer, v ]
    v = u2+w+2
END_PROVIDER

BEGIN_PROVIDER [ integer, u1 ]
    integer :: fu
! u1 = fu(d1,d2)
    u1 = d1+d2+1
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
    integer :: fu
! u2 = fu(d3,d4)
    u2 = d3+d4+1
    ASSERT (u2 > d3)
END_PROVIDER
integer function fu(x,y)
  integer :: x,y
  fu = x+y+1
end function

program irp_example
  print *, 't = ', t
end
Features

Arrays

```
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
Documentation

Every subroutine/function/provider can have a documentation section:

```
BEGIN_PROVIDER [ double precision, Fock_matrix_beta_mo, (mo_tot_num_align, mo_tot_num) ]
implicit none
BEGIN_DOC
! Fock matrix on the MO basis
END_DOC
...
END_PROVIDER
```

```
$ irpman fock_matrix_beta_mo
```
IRPF90 entities(l)          fock_matrix_beta_mo          IRPF90 entities(l)

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
Iterative processes
Iterative processes may involve cyclic dependencies:

TOUCH A: A is valid, but everything that needs A is invalidated
Meta-programming

BEGIN_SHELL [ /bin/bash ]
    echo print *, "Compiled by `whoami` on `date`"
    echo print *, "$FC $FCFLAGS"
    echo print *, "$IRPF90"
END_SHELL

BEGIN_SHELL [ /usr/bin/python ]
    for i in range(10):
        print ""
            double precision function times_%d(x)
                double precision, intent(in) :: x
                times_%d = x*%d
            end
        """%locals()
END_SHELL
Many Other features

- Color highlighting in Vi
- Generation of tags to navigate in the code
- Variables can be declared anywhere
- Dependencies are known by IRPF90 -> Makefiles are built automatically
- No problem using external libraries (MKL, MPI, etc)
- Compatible with OpenMP
- Support for Coarray Fortran (distributed parallelism)
- Codelet generation for code optimization
- Generation of Intel Fortran compiler directives to align arrays
- Generated code is very efficient (960 Tflops/s on Curie in 2011 with QMC=Chem)
- etc...