IRPF90, un generateur de code FORTRAN pour le calcul scientifique

Anthony Scemama

IRPF90: un générateur de code pour le calcul scientifique

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
Laboratoire de Chimie et Physique Quantiques
CNRS-Université de Toulouse

Aujourd'hui les gros codes scientifiques en Fortran deviennent difficiles à maintenir. La complexité des programmes provient des dépendences entre entités du code. Plus les entités sont inter-dépendantes, plus le programme est complexe. Pour que le code reste sous contrôle, il faut que le programmeur puisse connaître les conséquences d'une modification du code source sur tous les chemins d'exécution qui passent par le code modifié. Lorsque le code a été écrit par plusieurs développeurs et qu'il comporte des centaines de milliers de lignes, cela devient extrêmement difficile pour le programmeur. En revanche, la machine est tout-à-fait capable de faire ce travail à notre place en gérant les dépendances entre variables à la manière d'un Makefile.

IRPF90 est un générateur de code Fortran. De façon schématique, l'utilisateur n'écrit que des noyaux de calcul et IRPF90 génère le code qui va faire la liaison entre ces noyaux pour produire le résultat attendu en tenant compte des relations de dépendance entre les variables du programme. Ainsi, même les gros codes demeurent entièrement sous contrôle.

1 Présentation d'IRPF90

Un programme (ou sous-programme) scientifique est une fonction complexe de ses données. On peut représenter le programme comme un arbre dont la racine est la sortie du programme, les feuilles sont les données, les noeuds sont les variables intermédiaires et les segments représentent la relation *a besoin de*. Par exemple, le programme qui calcule *t(u(d1,d2), v(u(d3,d4), w(d5)))* avec

```
u(x)   = x + y + 1
v(x)   = x + y + 2
w(x)   = x + 3
t(x,y) = x + y + 4
```

peut être représenté par l'arbre suivant :

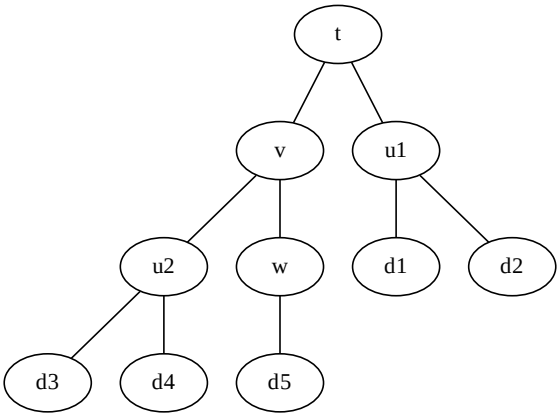

Figure 1: Arbre correspondant à la fonction t(u(d1,d2), v(u(d3,d4), w(d5)))

L'écriture du programme en Fortran nécessiterait au programmeur d'avoir cet arbre en tête :

```
program calcule_t
  implicit none
  integer :: d1, d2, d3, d4 d5  ! Donnees d'input
  integer :: u1, u2, v, w, t    ! Variables intermediaires
  call lecture(d1,d2,d3,d4,d5)
  call calcule_u(d1,d2,u1)
  call calcule_u(d3,d4,u2)
  call calcule_w(d5,w)
  call calcule_v(u2,w,v)
  call calcule_t(u1,v,t)
  write(*,*), "t=", t
end program
```

De cette façon, le programmeur dit à la machine ce qu'elle doit faire étape par étape : on parle de programmation impérative. Si les étapes ne sont pas données dans la bonne séquence, le programme est faux. Ainsi à chaque ligne, il faut connaître l'état de l'ensemble du programme, et donc les dépendence entre les variables. Dans cette approche, la pensée du programmeur va des feuilles de l'arbre vers la racine.

Le même programme peut être ré-écrit en pensant différemment. Au lieu de dire à la machine ce qu'elle doit faire étape par étape, on peut plutôt lui dire ce que l'on veut. Cela revient à penser le programme en partant de la racine vers les feuilles.

```
program calcule_t
  implicit none
  integer :: d1, d2, d3, d4 d5    ! Donnees d'input
  integer :: u1, u2, v, w, t      ! Fonctions
  call lecture(d1,d2,d3,d4,d5)
  write(*,*), "t=", t( u(d1,d2), v( u(d3,d4), w(d5) ) )
end program
```

Ici, les relations de dépendences entre les variables sont exprimées à travers l'appel de la fonction *t*, et le programmeur n'a plus besoin de spécifier une séquence particulière. Il ne connaît d'ailleurs pas *a priori* l'ordre dans lequel vont s'effectuer les appels des fonctions *w(d5)* et *u(d3,d4)*, mais il doit toujours avoir l'arbre en tête pour écrire le programme.

Avec IRPF90, le programmeur n'a pas besoin de connaître l'arbre : il est automatiquement calculé. Ainsi, avec IRPF90 le programme serait tout simplement :

```
program calcule_t
  write(*,*) t
end program

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

BEGIN_PROVIDER [ integer, u1 ]
 call calcule_u(d1,d2,u1)
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
 call calcule_u(d3,d4,u2)
END_PROVIDER

subroutine calcule_u(x,y,u)
 integer, intent(in)  :: x,y
 integer, intent(out) :: u
 u = x+y+1
end

BEGIN_PROVIDER  [ integer, d1 ]
&BEGIN_PROVIDER [ integer, d2 ]
&BEGIN_PROVIDER [ integer, d3 ]
&BEGIN_PROVIDER [ integer, d4 ]
&BEGIN_PROVIDER [ integer, d5 ]
 call lecture(d1,d2,d3,d4,d5)
END_PROVIDER
```

De cette façon, le programmeur exprime facilement sa pensée: "Imprime *t* à l'écran." Il n'a absolument pas besoin de savoir de quoi dépend *t*, comment *t* est calculé ou si *t* a déjà été calculé précédemment. Le programmeur veut *t* et rien de plus. Cela rend le développement collaboratif beaucoup plus simple.

Pour chaque noeud de l'arbre on écrit un **provider**, c'est-à-dire une subroutine dont le rôle est de construire la variable associée au noeud. Il est absolument nécessaire que la quantité soit contruite correctement dans le provider, tel que lorsqu'un provider est executé on ait la garantie que la quantité est construite correctement. Il est possible d'ajouter des assertions qui seront vérifées à l'exécution avec le mot-clé *ASSERT*.

```
BEGIN_PROVIDER [ integer, u2 ]
 call calcule_u(d3,d4,u2)
 ASSERT (u2 < d3)
END_PROVIDER
```

donne le résultat suivant en cas d'échec :

```
Stack trace:            0
-------------------------
provide_t
provide_v
provide_u2
u2
-------------------------
u2: Assert failed:
 file: uvwt.irp.f, line: 23
(u2 < d3)
u2 =            8
```

```
d3 =             3

STOP 1
```

IRPF90 analyse le code écrit dans tous les fichiers **.irp.f* du répertoire courant et repère les dépendances entre les variables. On voit dans le code que le provider de *v* a besoin de *u2* et de *w*. Ainsi, IRPF90 garantit qu'avant d'exécuter le provider de *v*, les providers de *u2* et de *w* auront été exécutés, et donc que les variables *u2* et *w* seront *valides*. Cependant, le programmeur ne sait pas à quel moment exact le provider de tel ou tel noeud sera appelé. L'utilisation de *t* dans le programme principal déclenche l'exploration récursive de l'arbre avant l'impression de la valeur de *t* à l'écran. Cela revient exactement à utiliser la fonction *t(u(d1,d2), v(u(d3,d4), w(d5)))* où les paramètres des fonctions sont implicites (Implicit Reference to Parameters in Fortran 90 : IRPF90).

Dès qu'un provider a été exécuté, la variable associée est marquée comme valide. Elle ne sera donc pas reconstruite mais tout simplement ré-utilisée si un autre provider a besoin de la même variable.

1.1 Compilation

IRPF90 est un générateur de code Fortran 90. La gestion de l'arbre est réalisée avant la compilation. Ainsi, IRPF90 génère un code Fortran90 où l'exploration de l'arbre est écrite directement dans le code généré. Ainsi, toute la gestion de l'arbre est statique et ne nuit absolument pas à la vitesse d'exécution du programme qui reste du Fortran90 standard. L'arbre est re-calculé avant chaque compilation car la modification du code peut induire de nouvelles dépendances entre les variables. Notons qu'il est bien entendu possible d'utiliser toutes les bibliothèques compatibles avec du Fortran (MPI, openMP, BLAS/Lapack, etc).

Il est possible d'écrire plusieurs programmes dans le même répertoire qui utilisent des providers communs. Cela est très utile pour construire des tests unitaires. Si l'on veut écrire un test pour la variable *u1*, il suffit d'écrire un programme principal qui imprime *u1* à l'écran, et seul le sous-arbre de *u1* sera généré à l'exécution.

Puisque IRPF90 connaît les dépendances entre les variables, il connaît également les dépendances entre les fichiers et peut donc écrire un Makefile automatiquement qui permettra de compiler tous les programmes du répertoire courant.

1.2 Modification dynamique des valeurs des noeuds

Les programmes scientifiques utilisent souvent des processus itératifs. Ceux-ci utilisent le même arbre de production à chaque itération, mais les valeurs des feuilles de l'itération *n+1* dépendent de la valeur de la racine de l'itération *n*. Cela implique de pouvoir modifier la valeur d'une variable à l'extérieur de son provider, et d'informer le système de cette modification afin que les dépendances entre variables soient mises à jour et que la nouvelle racine soit construite correctement.

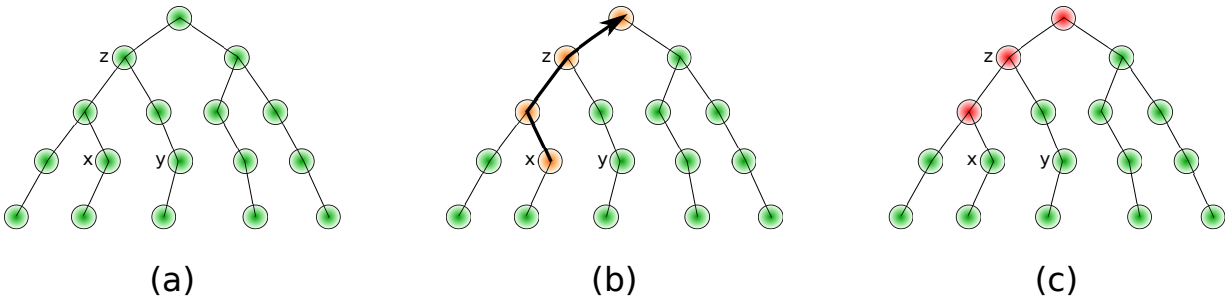

Figure 2: Invalidation de noeuds suite à l'utilisation de TOUCH x.

Le mot-clé *TOUCH* a été introduit pour gérer les modifications en dehors des providers. Ce mot-clé rend valide la variable modifiée (touchée), mais invalide tous les noeuds qui ont besoin directement ou

indirectement de cette variable. Dans l'exemple de la Figure 2, la variable *z* a besoin de *x* et de *y* pour être construite. En (a), l'arbre est dans un état où tout est construit et valide : tous les noeuds sont représentés en vert. En (b), on modifie la valeur de *x*, et ont exécute *TOUCH x*. En (c), on voit que *x* est valide, mais tous les noeuds entre lui et la racine ont été invalidés (en rouge). Ainsi, si l'on redemande la valeur de *z*, seul le noeud en rouge au dessous de *z* sera recalculé pour construire la nouvelle valeur de *z*.

Un exemple intéressant du *TOUCH* est le calcul d'une dérivée par différence finie, qui peut être utilisé pour vérifier l'implémentation de la dérivée d'une fonction. L'exemple suivant calcule la dérivée de *F* par rapport à *x* :

```
BEGIN_PROVIDER [ real, dF ]
  real :: F_p, F_m
  real, parameter :: delta_x = 0.001

  ! Calcul de F(x + 1/2.delta_x)
  x += 0.5*delta_x
  TOUCH x
  F_p = F

  ! Calcul de F(x - 1/2.delta_x)
  x -= delta_x
  TOUCH x
  F_m = F

  ! Calcul de dF
  dF = (F_p - F_m)/delta_x

  ! On remet x a sa position initiale
  x += 0.5*delta_x
  TOUCH x
END_PROVIDER
```

1.3 Variables tableaux

Un tableau est considéré comme valide lorsque toutes ses valeurs ont été calculées. Les dimensions des tableaux sont soit des variables qui ont des providers, soit des constantes, soit des intervalles.

```
BEGIN_PROVIDER [ integer, fact_max ]
  fact_max = 10
END_PROVIDER

BEGIN_PROVIDER [ double precision, fact, (0:fact_max) ]
  implicit none
  integer :: i
  fact(0) = 1.d0
  do i=1,fact_max
    fact(i) = fact(i-1)*dble(i)
  end do
END_PROVIDER
```

Dans cet exemple, puisque le tableau *fact* dépend de sa variable de dimensionnement *fact_max*, la modification de la dimension du tableau à travers *TOUCH fact_max* invalidera le tableau *fact*, et il sera re-calculé avec la bonne dimension à sa prochaine utilisation. Toutes les allocations sont vérifiées et un message d'erreur apparaît à l'exécution en cas d'impossibilité d'allocation du tableau.

La mémoire réservée pour un tableau peut être libérée en utilisant le mot-clé *FREE*. Par exemple :

```
BEGIN_PROVIDER [ double precision, table2, (size(table1,1)) ]
  implicit none
  table2(:) = 2.d0 * table1(:)
  FREE table1
END_PROVIDER
```

Lorsque *table1* est libéré, le noeud correspondant est marqué comme invalide. Ainsi, si *table1* est re-demandé plus tard, il sera préalablement ré-alloué et reconstruit.

1.4 Documentation du code

À l'intérieur de chaque provider, il est recommandé d'écrire quelques lignes de documentation pour décrire la variable construite.

```
BEGIN_PROVIDER [ double precision, fact, (0:fact_max) ]
  implicit none

  BEGIN_DOC
!  Computes an array of fact(n)
  END_DOC

  integer :: i
  fact(0) = 1.d0
  do i=1,fact_max
    fact(i) = fact(i-1)*dble(i)
  end do
END_PROVIDER
```

Lors de la compilation, la liste de toutes les variables est écrite dans un fichier nommé *irpf90_entities*, et une *man page* est crée pour chaque variable. Cette page contient la documentation présente dans le block de documentation, mais aussi quelles sont les variables nécessaires et quelles variables ont besoin de la variables courante. Ces pages sont accessibles avec l'outil *irpman* :

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

Declaration
       double precision, allocatable :: fact   (0:fact_max)

Description
       !  Computes an array of fact(n)

File
       fact.irp.f

Needs
       fact_max

Needed by
       binom
       exponential_series

IRPF90 entities                           fact                        IRPF90 entities(l)
```

1.5 Templates

Il arrive parfois que l'on doive écrire plusieurs morceaux de code qui utilisent un même schéma. En C++ par exemple on utiliserait des patrons de fonctions, de classes ou d'expressions. Prenons un exemple où l'on veut créer plusieurs providers et fonctions semblables :

```
BEGIN_TEMPLATE

  BEGIN_PROVIDER [ $type , $name ]
   call find_in_input('$name', $name)
  END_PROVIDER

  logical function $name_is_zero()
    $name_is_zero = ($name == 0)
  end function

SUBST [ type, name ]

  integer    ;   size_tab1 ;;
  integer    ;   size_tab2 ;;
  real       ;   distance  ;;
  real       ;   x         ;;
  real       ;   y         ;;
  real       ;   z         ;;

END_TEMPLATE
```

Cet exemple génère automatiquement un provider et une fonction **_is_zero* pour chaque couple apparaissant au dessous du mot-clé *SUBST*.

1.6 Interaction avec le shell et metaprogrammation

On peut avoir envie d'insérer au milieu de son programme le résultat d'une commande du shell exécutée à la compilation. Prenons un cas typique où l'on souhaite que le programme imprime à l'écran la date de compilation et la version de git à laquelle il correspond :

```
subroutine print_git_log
 write(*,*) , '----------------'
 BEGIN_SHELL [ /bin/bash ]
    echo "write(*,*)" \'Compiled by $(whoami) on $(date)\'
 END_SHELL
 write(*,*) , 'Last git commit:'
 BEGIN_SHELL [ /bin/bash ]
   git log -1 | sed "s/'//g" | sed "s/^/    write(*,*) '/g" | sed "s/$/'/g"
 END_SHELL
 write(*,*) , '----------------'
end
```

L'insertion de sorties de scripts dans le code permet également d'aller au delà des templates, et de générer du code de façon mécanique. Voici un exemple qui génère des fonctions particulières pour calculer *x* à la puissance *n* en utilisant un minimum de multiplications :

```
BEGIN_SHELL [ /usr/bin/python ]
```

```
POWER_MAX = 20

def compute_x_prod(n,d):
  if n == 0:
    d[0] = None
    return d
  if n == 1:
    d[1] = None
    return d
  if n in d:
    return d
  m = n/2
  d = compute_x_prod(m,d)
  d[n] = None
  d[2*m] = None
  return d

def print_function(n):
  keys = compute_x_prod(n,{}).keys()
  keys.sort()
  output = []
  print "real function power_%d(x1)"%n
  print " real, intent(in) :: x1"
  for i in range(1,len(keys)):
    output.append( "x%d"%keys[i] )
  print " real :: "+', '.join(output)
  for i in range(1,len(keys)):
    ki = keys[i]
    ki1 = keys[i-1]
    if ki == 2*ki1:
      print " x%d"%ki + " = x%d * x%d"%(ki1,ki1)
    else:
      print " x%d"%ki + " = x%d * x1"%(ki1)
  print " power_%d = x%d"%(n,n)
  print "end"

for i in range(POWER_MAX):
  print_function(i+1)
  print ''

END_SHELL
```

Voici un échantillon du code généré :

```
real function power_1(x1)
 real, intent(in) :: x1
 real ::
 power_1 = x1
end

real function power_2(x1)
 real, intent(in) :: x1
 real :: x2
 x2 = x1 * x1
```

```
 power_2 = x2
end

   ...

real function power_20(x1)
 real, intent(in) :: x1
 real :: x2, x4, x5, x10, x20
 x2 = x1 * x1
 x4 = x2 * x2
 x5 = x4 * x1
 x10 = x5 * x5
 x20 = x10 * x10
 power_20 = x20
end
```

2 IRPF90 pour le HPC

2.1 Vectorisation

IRPF90 offre certaines possibilités pour améliorer facilement les performances des programmes. Il peut générer automatiquement des directives du compilateur Intel pour aider la vectorisation. En effet, pour que le compilateur génère une instruction vectorielle, il est nécessaire que les opérandes soient alignées sur une frontière de 16, 32, ou 64 octets selon le jeu d'instructions. Une boucle Fortran traditionnelle va générer trois boucles :

- une boucle d'"épluchage" (peel loop)
- une boucle vectorielle
- une boucle de reste (tail loop)

La peel loop est une boucle qui traite les élement de départ jusqu'à la frontière d'alignement, et la boucle de reste traite les derniers élement.

Prenons la boucle suivante que l'on voudra vectoriser avec des instructions AVX (32 octets), où *nmax* = 26, et le tableau *a* est de type double précision.

```
double precision :: a(nmax)
do i=1,nmax
  a(i) = 2. * a(i)
end do
```

La boucle scalaire va faire 26 tours. Si l'alignement du tableau est tel que le premier élément du tableau est à 8 octets après la frontière, la peel loop va effectuer trois tours pour atteindre la frontière de 32 octets. Puis, la boucle vectorisée va prendre cinq fois les éléments suivants de *a* quatre par quatre, jusqu'au 23ième élement inclus. Ensuite, la boucle de reste va traiter les trois éléments restants. Avec la vectorisation, le nombre de tours devient 3+5+3=11 tours. Dans ce cas, la vectorisation fait gagner un facteur 2.36.

Si maintenant le tableau est aligné sur 32 octets avec la directive suivante (pour le compilateur Intel) :

```
!DIR$ ATTRIBUTES ALIGN : 32 :: a
```

la peel loop ne sera pas générée, la boucle vectorielle fera 6 tours, et les éléments 25 et 26 seront traités avec la boucle de reste. On aura don 6+2=8 tours, ce qui correspond a un gain de 3.25.

Enfin, on peut aussi forcer la vectorisation complète de la boucle de la façon suivante :

```
double precision :: a(nmax+4)
!DIR$ ATTRIBUTES ALIGN : 32 :: a

do i=1, 1+(nmax-1)/4, 4
  !DIR$ VECTOR ALIGNED
  a(i:i+3) = 2. * a(i:i+3)
end do
```

Ici, les boucles implicites *a(i:i+3)* seront vectorisées, et on aura au total 7 tours, soit un gain de 3.71.

Cet exemple montre deux aspects importants (pour les boucles qui font peu de tours) :

1. L'alignement des tableaux ne peut qu'améliorer l'efficacité de la vectorisation
2. Pour pouvoir vectoriser à 100% une boucle, il faut connaître la largeur des vecteurs pour les instructions vectorielles (16 octets pour SSE, 32 octets pour AVX, etc), et donc écrire un code spécifique pour l'architecture visée.

Il est possible de demander à IRPF90 de générer les directives d'alignement pour *toutes* les variables tableaux en utilisant par exemple :

```
irpf90 --align=32
```

IRPF90 va également chercher dans tout le code la variable *$IRP_ALIGN* et la substituer par 32. Ainsi on peut écrire le code précédent comme:

```
do i=1, 1+(nmax-1)/$IRP_ALIGN, $IRP_ALIGN
  !DIR$ VECTOR ALIGNED
  a(i:i+$IRP_ALIGN-1) = 2. * a(i:i+$IRP_ALIGN-1)
end do
```

afin d'avoir un code source portable qui s'adapte à l'architecture visée.

2.2 Génération de code spécifique

Plus le compilateur Fortran a d'informations, plus il peut produire un code efficace. IRPF90 possède une option qui permet de substituer des variables du programme par des valeurs dans les bornes des boucles et dans les conditions (if). Ce type d'optimisation fait gagner en général entre 5 et 15% sur un vrai code, mais bien entendu le programme donnera des résultats faux si les données ne correspondent pas a ce qui a été spécifié à la compilation.

Reprenons la boucle scalaire de l'exemple précédent :

```
double precision :: a(nmax)
!DIR$ ATTRIBUTES ALIGN : $IRP_ALIGN :: a

if (calculate_loop) then
  do i=1,nmax
    a(i) = 2. * a(i)
  end do
end if
```

et utilisons :

```
irpf90 --align=32 --substitute nmax:26 --substitute calculate_loop:.True.
```

le code produit sera

```
double precision :: a(nmax)
!DIR$ ATTRIBUTES ALIGN : 32 :: a

if (.True.) then
  do i=1,26
    a(i) = 2. * a(i)
  end do
end if
```

Le compilateur Fortran va supprimer le test qui est toujours vrai, et produire le binaire optimal pour une boucle a 26 tours.

Si l'on utilise :

```
irpf90 --align=32 --substitute nmax:26 --substitute calculate_loop:.False.
```

la condition étant toujours fausse le compilateur Fortran va supprimer tout le code contenu dans le if.

2.3 Profiling

Quand on fait développement pour le HPC, il est absolument nécessaire de pouvoir isoler la portion de code que l'on cherche à optimiser. Étant donné que l'arbre du programme est géré par IRPF90, il peut générer automatiquement des codelets pour chacun des noeuds de l'arbre :

irpf90 --codelet=tab2:10000

va construire un programme principal qui va appeler le provider de *tab2* 10000 fois, et mesurer le nombre de cycles et le temps en secondes nécessaire pour construire *tab2*. Si l'on souhaite profiler tout le code, il suffit d'ajouter l'option *-g* et un rapport de profiling appraîtra dans la sortie standard après l'exécution du programme.

3 En bref

Écrire un programme revient à écrire les providers des variables impliquées. Lorsqu'on écrit un provider pour une variable *x*, les seules questions que l'on se pose sont :

- Comment est-ce que je construis *x* ?
- Quelles sont les noms des variables dont j'ai besoin ?
- Suis-je bien sûr que *x* est valide à la sortie du provider ?

La programmation quotidienne est facilitée :

- La séquence d'exécution n'a pas besoin d'être connue
- Les makefiles sont construits automatiquement
- Lorsqu'on utilise une quantité, on est absolument sûr qu'elle est a déjà été construite correctement
- Écrire un nouveau provider ne va jamais nuire au reste du code: on ne risque pas de "casser" un programme qui fonctionnait
- Plusieurs personnes peuvent travailler simultanément sur le même code avec un minimum d'effort

- Si une variable a déjà été construite, elle ne sera jamais re-construite si cela n'est pas nécessaire.
- La construction de tests unitaires et codelets pour le travail d'optimisation consiste simplement à créer un nouveau programme principal.

IRPF90 est un programme en GPL téléchargeable ici: http://irpf90.ups-tlse.fr. Ce site comporte plusieurs tutoriels et présentations.