

Présentation de l'outil IRPF90

Anthony Scemama

LCPQ-IRSAMC, Toulouse
scemama@irsamc.ups-tlse.fr
<http://irpf90.ups-tlse.fr>

31 Mai 2011

Présentation

- 1 Introduction
- 2 Programmation par IRPF90
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90
- 5 Conclusion

Préambule

- Un programme (ou sous-programme) est une fonction des données :
$$\text{output} = \text{programme}(\text{input})$$
- Dans la vision fonctionnelle, on peut toujours représenter un programme comme un graphe sans cycle où :
 - Les nœuds du graphe sont les variables
 - Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
 - On parle d'*arbre de production*

Préambule

- Un programme (ou sous-programme) est une fonction des données :
$$\text{output} = \text{programme}(\text{input})$$
- Dans la vision fonctionnelle, on peut toujours représenter un programme comme un graphe sans cycle où :
 - Les nœuds du graphe sont les variables
 - Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
 - On parle d'*arbre de production*

Préambule

- Un programme (ou sous-programme) est une fonction des données :
$$\text{output} = \text{programme}(\text{input})$$
- Dans la vision fonctionnelle, on peut toujours représenter un programme comme un graphe sans cycle où :
 - Les nœuds du graphe sont les variables
 - Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
 - On parle d'*arbre de production*

Préambule

- Un programme (ou sous-programme) est une fonction des données :
$$\text{output} = \text{programme}(\text{input})$$
- Dans la vision fonctionnelle, on peut toujours représenter un programme comme un graphe sans cycle où :
 - Les nœuds du graphe sont les variables
 - Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
 - On parle d'*arbre de production*

Préambule

- Un programme (ou sous-programme) est une fonction des données :
$$\text{output} = \text{programme}(\text{input})$$
- Dans la vision fonctionnelle, on peut toujours représenter un programme comme un graphe sans cycle où :
 - Les nœuds du graphe sont les variables
 - Les segments représentent la relation “a besoin de” ou “est nécessaire pour” (“est un argument de”).
 - On parle d'*arbre de production*

Exemple

On cherche à calculer $t(u(d_1, d_2), v(u(d_3, d_4), w(d_5)))$:

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

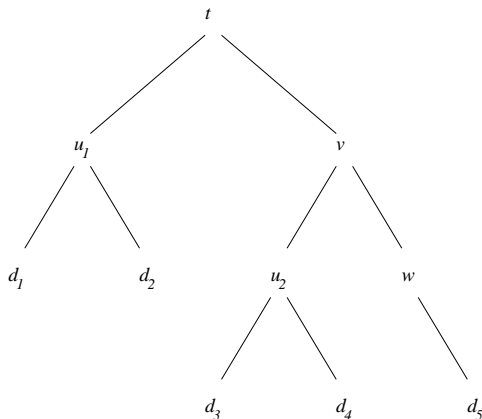
$$v(x, y) = x + y + 2 \quad (2)$$

$$w(x) = x + 3 \quad (3)$$

$$t(x, y) = x + y + 4 \quad (4)$$

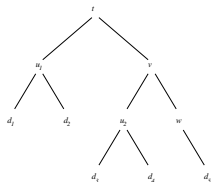
Exemple

Arbre de production de t :



Programmation usuelle

```
program exemple_1
  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)
  print *, t
end program
```



Autres possibilités

```
program exemple_2
  call lecture(d1,d2,d3,d4,d5)
  u1 = calcule_u(d1,d2)
  u2 = calcule_u(d3,d4)
  w = calcule_w(d5)
  v = calcule_v(u2,w)
  t = calcule_t(u1,v)
  print *, t
end program
```

ou même :

```
program exemple3
  call lecture(d1,d2,d3,d4,d5)
  print *, t( u(d1,d2) , v( u(d3,d4),w(d5) ) )
end program
```

Autres possibilités

```
program exemple_2
  call lecture(d1,d2,d3,d4,d5)
  u1 = calcule_u(d1,d2)
  u2 = calcule_u(d3,d4)
  w = calcule_w(d5)
  v = calcule_v(u2,w)
  t = calcule_t(u1,v)
  print *, t
end program
```

ou même :

```
program exemple3
  call lecture(d1,d2,d3,d4,d5)
  print *, t( u(d1,d2) , v( u(d3,d4),w(d5) ) )
end program
```

Présentation

- 1 Introduction
- 2 Programmation par IRPF90**
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90
- 5 Conclusion

Programmation par IRP90

fichier: exemple.irp.f

```
program exemple_4
  print *, t
end program
```

- L'utilisation de *t* engendre l'exploration de l'arbre.
- Équivalent à un appel de fonction mais sans paramètres (IRP : Implicit Reference to Parameters)

Programmation par IRP90

fichier: exemple.irp.f

```
program exemple_4  
  print *, t  
end program
```

- L'utilisation de *t* engendre l'exploration de l'arbre.
- Équivalent à un appel de fonction mais sans paramètres (IRP : Implicit Reference to Parameters)

Programmation par IRP90

fichier: `exemple.irp.f`

```
program exemple_4
  print *, t
end program
```

- L'utilisation de *t* engendre l'exploration de l'arbre.
- Équivalent à un appel de fonction mais sans paramètres (IRP : Implicit Reference to Parameters)

Construction des nœuds de l'arbre

On écrit comment construire un nœud en fonction de ses fils

fichier: uvwt.irp.f

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

```
BEGIN_PROVIDER [ integer, u1 ]  
    u1 = d1+d2+1  
END_PROVIDER
```

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

Construction des feuilles de l'arbre

On peut aussi écrire un seul “provider” pour plusieurs variables :

fichier: input.irp.f

```
BEGIN_PROVIDER [ integer, d1 ]
&BEGIN_PROVIDER [ integer, d2 ]
&BEGIN_PROVIDER [ integer, d3 ]
&BEGIN_PROVIDER [ integer, d4 ]
&BEGIN_PROVIDER [ integer, d5 ]
  read(*,*) d1
  read(*,*) d2
  read(*,*) d3
  read(*,*) d4
  read(*,*) d5
END_PROVIDER
```

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
`BEGIN_PROVIDER ... END_PROVIDER`
- Des appels `call provide_*` sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
`BEGIN_PROVIDER ... END_PROVIDER`
- Des appels `call provide_*` sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
`BEGIN_PROVIDER ... END_PROVIDER`
- Des appels `call provide_*` sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
`BEGIN_PROVIDER ... END_PROVIDER`
- Des appels `call provide_*` sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
BEGIN_PROVIDER ... END_PROVIDER
- Des appels call provide_* sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
BEGIN_PROVIDER ... END_PROVIDER
- Des appels call provide_* sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
BEGIN_PROVIDER ... END_PROVIDER
- Des appels call provide_* sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Génération du Fortran

- Le programme *irpf90* lit tous les fichiers *.irp.f du répertoire courant
- Toutes les entités IRP sont répertoriées
- Toutes les entités IRP sont recherchées dans le code
- Les providers sont créés à partir des blocs
BEGIN_PROVIDER ... END_PROVIDER
- Des appels call provide_* sont insérés dans le texte
- Chaque fichier *.irp.f engendre un fichier *.F90 contenant un module *_mod.
- Un module contient les entités IRP dans le fichier, et l'information de la validité des données.
- Un fichier contenant les dépendances entre modules est construit pour le Makefile

Présentation

- 1 Introduction
- 2 Programmation par IRPF90
- 3 Fonctionnement interne de IRPF90**
- 4 Possibilités de IRPF90
- 5 Conclusion

Fortran généré

fichier: exemple_4.irp.F90

```
program exemple_4
  use uvwt_mod      ! Module qui contient t
  call provide_t    ! Fournit (et construit) t
  print *, t        ! Code du fichier exemple_4.irp.f
end
```

fichier: input.irp.module.F90

```
module input_mod
  integer :: d1
  logical :: d1_is_built = .False.
  integer :: d5,d4,d3,d2
end module input_mod
```

fichier: input.irp.F90

```
subroutine provide_d1
  use input_mod
  if (.not.d1_is_built) then
    d1_is_built = .True.
    call bld_d1
  endif
end subroutine provide_d1
...
```

fichier: input.irp.F90

```
...  
subroutine bld_d1  
  use input_mod      ! Code insere  
  read(*,*) d1  
  read(*,*) d2  
  read(*,*) d3  
  read(*,*) d4  
  read(*,*) d5  
end subroutine bld_d1  
...
```

fichier: uvwt.module.irp.F90

```
module uvwt_mod
  integer :: u2
  logical :: u2_is_built = .False.
  integer :: u1
  logical :: u1_is_built = .False.
  integer :: v
  logical :: v_is_built = .False.
  integer :: w
  logical :: w_is_built = .False.
  integer :: t
  logical :: t_is_built = .False.
end module uvwt_mod
```

fichier: uvwt.irp.F90

```
subroutine provide_u2
  use input_mod
  use uvwt_mod
  if (.not.u2_is_built) then
    call provide_d1           ! Fournit aussi d2,d3,d4,d5
    u2_is_built = .True.
    call bld_u2
  endif
end subroutine provide_u2
...
```


fichier: uvwt.irp.F90

```
...  
subroutine bld_u2  
  use input_mod  
  use uvwt_mod  
  integer :: fu  
  u2 = fu(d3,d4)  
end subroutine bld_u2  
...
```

On peut utiliser des sous-routines et des fonctions Fortran, comme la fonction `fu` dans cet exemple.

Présentation

- 1 Introduction
- 2 Programmation par IRPF90
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90**
- 5 Conclusion

Déclaration des tableaux

Par exemple, un tableau A est déclaré par :

```
BEGIN_PROVIDER [ integer, A, (dim1,dim2) ]
```

où dim1 et dim2 peuvent être :

- des entiers
- des entités IRP
- un domaine a:b entre a et b

Déclaration des tableaux

Par exemple, un tableau A est déclaré par :

```
BEGIN_PROVIDER [ integer, A, (dim1,dim2) ]
```

où dim1 et dim2 peuvent être :

- des entiers
- des entités IRP
- un domaine a:b entre a et b

Déclaration des tableaux

Par exemple, un tableau A est déclaré par :

```
BEGIN_PROVIDER [ integer, A, (dim1,dim2) ]
```

où dim1 et dim2 peuvent être :

- des entiers
- des entités IRP
- un domaine a:b entre a et b

Dans la routine `provide_a`, si la variable `A` n'est pas construite :

- Si le tableau `A` n'est pas encore alloué, on fait l'allocation.
- Si le tableau `A` est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des entités IRP, on appelle les providers de ces variables avant de procéder à l'allocation.

Dans la routine `provide_a`, si la variable `A` n'est pas construite :

- Si le tableau `A` n'est pas encore alloué, on fait l'allocation.
- Si le tableau `A` est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des entités IRP, on appelle les providers de ces variables avant de procéder à l'allocation.

Dans la routine `provide_a`, si la variable A n'est pas construite :

- Si le tableau A n'est pas encore alloué, on fait l'allocation.
- Si le tableau A est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des entités IRP, on appelle les providers de ces variables avant de procéder à l'allocation.

Dans la routine `provide_a`, si la variable `A` n'est pas construite :

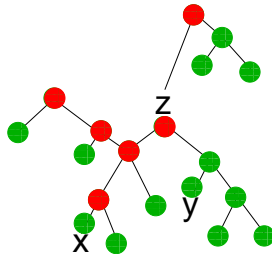
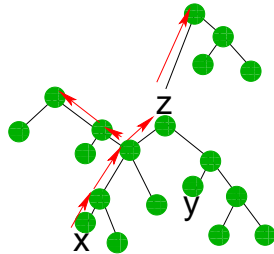
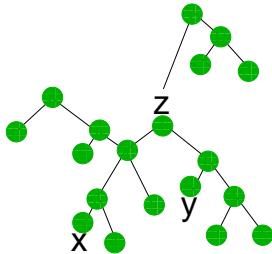
- Si le tableau `A` n'est pas encore alloué, on fait l'allocation.
- Si le tableau `A` est déjà alloué, on vérifie que ses dimensions sont correctes. Si elles ne le sont pas, on le ré-alloue avec les bonnes dimensions.
- En cas d'allocation, on vérifie que l'allocation s'est bien produite (`stat=err`)
- Si les dimensions sont des entités `IRP`, on appelle les `providers` de ces variables avant de procéder à l'allocation.

Modification d'entités IRP

La modification d'une entité IRP entraîne l'invalidation de tous ses parents. Cela est réalisé avec le mot-clé TOUCH :

```
do i=1,10
  do j=1,10
    x(j) = x(j)+3
  enddo
  TOUCH x
  print *, y ! y est different si y depend de x
enddo
```

Si y et x ont un parent z en commun celui-ci sera donc recalculé, ainsi que tous les intermédiaires entre z et y



Assertions

Une assertion est exprimée par le mot-clé ASSERT

ASSERT (d2 > d1)

- Si l'option `-a` n'est pas présente dans la ligne de commande, les assertions sont supprimées
- Si la condition n'est pas remplie, un message d'erreur est produit à l'exécution, avec la pile d'exécution

Stack trace:

```
-----  
irp_example1  
subroutine run  
provide_t  
provide_v  
provide_w  
provide_d5  
bld_d1  
-----  
bld_d1: Assert failed:  
file: input.irp.f, line: 8  
( d3 > d2 )  
d3 =          2  
d2 =          3
```

Assertions

Une assertion est exprimée par le mot-clé ASSERT

ASSERT (d2 > d1)

- Si l'option `-a` n'est pas présente dans la ligne de commande, les assertions sont supprimées
- Si la condition n'est pas remplie, un message d'erreur est produit à l'exécution, avec la pile d'exécution

Stack trace:

```
-----  
irp_example1  
subroutine run  
provide_t  
provide_v  
provide_w  
provide_d5  
bld_d1  
-----  
bld_d1: Assert failed:  
file: input.irp.f, line: 8  
( d3 > d2 )  
d3 =          2  
d2 =          3
```

Assertions

Une assertion est exprimée par le mot-clé ASSERT

ASSERT (d2 > d1)

- Si l'option `-a` n'est pas présente dans la ligne de commande, les assertions sont supprimées
- Si la condition n'est pas remplie, un message d'erreur est produit à l'exécution, avec la pile d'exécution

Stack trace:

```
-----  
irp_example1  
subroutine run  
provide_t  
provide_v  
provide_w  
provide_d5  
bld_d1  
-----  
bld_d1: Assert failed:  
file: input.irp.f, line: 8  
( d3 > d2 )  
d3 =          2  
d2 =          3
```

Introduction de scripts dans le code

Il est possible d'introduire des scripts dont le résultat est du code fortran. Par exemple :

fichier: `exemple_5.irp.f`

```
program example_5
  BEGIN_SHELL [ /bin/bash ]
    echo print *, \'Compiled by \'whoami\' on \'date\'\'
  END_SHELL
end
```

donne :

fichier: `exemple_5.irp.F90`

```
program example_5
print *, 'Compiled by scemama on Fri Sep 5 09:14:22 CEST 2008'
end
```

Le bloc BEGIN_SHELL...END_SHELL peut aussi permettre de construire des blocs BEGIN_PROVIDER...END_PROVIDER :

```
BEGIN_SHELL [ /usr/bin/python ]  
for i in [1,2]:  
    var="x_"+str(i)  
    print "BEGIN_PROVIDER [ integer, "+var+" ]"  
    print " "+var+" = "+str(i)  
    print "END_PROVIDER"  
END_SHELL
```

```
BEGIN_PROVIDER [ integer, x_1 ]  
    x_1 = 1  
END_PROVIDER  
BEGIN_PROVIDER [ integer, x_2 ]  
    x_2 = 2  
END_PROVIDER
```


Compilation conditionnelle

Il est possible de définir des variables dans la ligne de commande avec l'option `-D`. Par exemple : `irpf90 -DMPI` définit la variable `MPI`.

Ces variables définissent le code fortran à générer en utilisant la construction `IRP_IF...IRP_ELSE...IRP_ENDIF`. Par exemple :

```
subroutine abort
  IRP_IF MPI
    include 'mpif.h'
    call mpi_abort(MPI_COMM_WORLD,0,ierr)
  IRP_ELSE
    stop
  IRP_ENDIF
end
```

Debug

Chaque subroutine, fonction et provider de variable contient la chaîne de caractères `irp_here` qui contient le nom de la routine.
L'ajout de

```
print *, irp_here
```

dans le code imprime le nom de la routine courante.

La présence de l'option `-d` dans la ligne de commande imprime (pour la routine `r`) :

```
-> r           Entrée dans la routine r
```

```
<- r   xxx s   Sortie de la routine r
```

ou `xxx` est le temps CPU passé dans `r`.

Présentation

- 1 Introduction
- 2 Programmation par IRPF90
- 3 Fonctionnement interne de IRPF90
- 4 Possibilités de IRPF90
- 5 Conclusion**

- Simplicité d'écriture du code
- Simplicité de débogage : la complexité reste faible
- Code généré très performant
- Parallélisation automatique possible
- Portabilité du code généré