Ada: réunir sécurité et performance

Séminaire « Performance et généricité »

Samuel Tardieu
sam@rfc1149.net

École Nationale Supérieure des Télécommunications

5 novembre 2008

Le langage Ada en une page

- Défini par une norme ISO
- Plusieurs variantes existent (Ada 83, Ada 95, Ada 2005)
- Fortement et statiquement typé
- Orienté objet (héritage simple et interfaces)
- Intègre le parallélisme et la répartition au cœur du langage
- ~~Très~~ Légèrement verbeux
- « Quand ça compile, ça marche »

Caractéristiques des exemples

Voici, pour pouvoir les reproduire, les caractéristiques du compilateur utilisé pour les exemples qui suivront.

- OS : GNU/Linux
- GCC 4.4.0 20081103 (experimental)
- Options de compilation : -O3 -fomit-frame-pointer
- Cible : x86_64 (64 bits)

Types et sous-types

Ada permet de créer de nouveaux types ou de sous-classer des types existants.

Exemple de déclarations

```
type Small_Integer is range 0 .. 31;
--  Prend toutes les valeurs entre 0 et 31.

subtype Birth_Year is Positive range 1800 .. 2100;
--  Sous-ensemble compatible avec Positive.

type Death_Year is new Positive range 1800 .. 2100;
--  Nouveau type incompatible avec Positive.

type My_Array is array (Positive range <>) of Boolean;
--  Tableau de booleens de taille inconnue.

subtype My_Other_Array is My_Array (1 .. 5);
--  Tableau de booleens compatible, de taille connue.

subtype Yet_Another_Array is My_Array (0 .. 5);
--  Erreur, l'index n'est pas compatible avec la contrainte.
```

Discriminants

Les types avec discriminant permettent d'avoir des variantes. Au contraire des unions en C, les objets connaissent leur variante, et il est impossible de modifier ou consulter des champs incorrects.

Type discriminé

```
type Command (Payed : Boolean) is record

   -- Partie commune.
   Number : Command_Number;

   -- Partie variante.
   case Payed is
      when True =>
         Cashed : Date;
      when False =>
         Due : Date;
   end case;
end record;

subtype Payed_Command is Command (Payed => True);
--  Ne peut stocker qu'une commande dont le payement a eu lieu.
```

Passage de paramètres

Il existe plusieurs modes de passage de paramètres à un sous-programme :

- **in** *(procédure, fonction) :* Paramètre en lecture seule.
- **out** *(procédure) :* Paramètre en écriture seule, variable temporaire à l'intérieur de la procédure.
- **in out** *(procédure) :* Paramètre en lecture-écriture.
- **access** *(procédure, fonction) :* Paramètre passé par référence (pointeur). Peut contenir des qualificatifs supplémentaires (**not null**, **constant**).

Le mode du paramètre a des implications sur les performances.

Un premier exemple

Considérons ce code (stupide) en C++. Le but est d'écrire une fonction qui renvoie un entier augmenté et diminué de 1.

C++

```
// Fonction generique, reutilisable, devrait faire l'objet d'une
// bibliotheque a elle toute seule (d'apres le patron).
void inc_dec(int x, int& i, int& d)
{
  i = x+1;
  d = x-1;
}

// Fonction de test.
int r()
{
  // Le patron interdit les variables non initialisees.
  int i = 0, d = 0;
  inc_dec(43, i, d);
  return d;
}
```

Code C++

Compilation du code C++

```
_Z7inc_deciRiS_:
        leal    1(%rdi), %eax
        subl    $1, %edi
        movl    %eax, (%rsi)
        movl    %edi, (%rdx)
        ret

_Z1rv:
        subq    $24, %rsp
        movl    $43, %edi
        leaq    8(%rsp), %rdx
        leaq    12(%rsp), %rsi
        movl    $0, 12(%rsp)
        movl    $0, 8(%rsp)
        call    _Z7inc_deciRiS_
        movl    8(%rsp), %eax
        addq    $24, %rsp
        ret
```

La même chose en Ada

Ada

```
procedure Inc_Dec (X : in Integer;
                   I : out Integer;
                   D : out Integer)
is
begin
   I := X + 1;
   D := X - 1;
end Inc_Dec;

function R return Integer is
   I, D : Integer := 0;   -- Variables non initialisees interdite par le patron.
begin
   Inc_Dec (43, I, D);
   return D;
end R;
```

Code Ada

Compilation du code Ada

```
q__inc_dec:
        leal    -1(%rdi), %eax
        addl    $1, %edi
        salq    $32, %rax
        orq     %rdi, %rax
        ret

p__r:
        subq    $8, %rsp
        movl    $43, %edi
        call    q__inc_dec
        sarq    $32, %rax
        addq    $8, %rsp
        ret
```

Analyse des différences

- En C++, **inc_dec** a le droit de lire ou de stocker les références I et D.
- Le code Ada sait que I et D ne seront pas lus.
- On obtiendra le même résultat en C++ qu'en Ada en créant une structure arbitraire regroupant les paramètres en sortie seule et en retournant cette structure.
- On obtiendra le même résultat en Ada qu'en C++ en utilisant des paramètres de type **not null access Integer** plutôt que **out Integer**.
- L'important ici est que l'écriture *naturelle* en Ada est la plus efficace.

Déplacement des vérifications

Vérification faite par l'appelé

```
procedure FQ (A : access Integer) is
begin
   A.all := 42;
end FQ;
```

Code expansé

```
procedure q__fq (a : access integer) is
begin
   [constraint_error when
     a = null
     "access check failed"]
   a.all := 42;
   return;
end q__fq;
```

Il est très mal vu de laisser le compilateur faire les vérifications spontanément lorsqu'on sait qu'on doit les faire.

Interlude

Notons que le code de vérification n'est inclus que s'il est nécessaire :

Vérification explicite

```
procedure FQ (A : access Integer) is
begin
   if A /= null then
      A.all := 42;
   end if;
end FQ;
```

Code expansé

```
procedure q__fq (a : access integer) is
begin
   if a /= null then
      a.all := 42;
   end if;
   return;
end q__fq;
```

Déplacement des vérifications (suite)

Il est plus raisonnable d'utiliser la contrainte **not null** dans la signature du sous-programme.

Vérification faite par l'appelant

```
procedure FQ (A : not null access Integer) is
begin
   A.all := 42;
end FQ;
```

Code expansé

```
procedure q__fq (a : not null access integer) is
begin
   a.all := 42;
   return;
end q__fq;
```

C'est l'appelant, s'il en a besoin, qui effectuera la vérification.

Expressions statiques et constantes nommées

Ada, comme la plupart des langages, permet d'effectuer certains calculs à la compilation. Lequel de ces deux codes est correct ?

Utilisation de constante nommée en Ada

```
function Const return Integer is
   C1 : constant := 2**100 / 3;   -- Grand et pas entier
   C2 : constant := 2**100 / 9;   -- Grand et pas entier non plus
begin
   return 30 * C1 / C2;           -- Ca marche ?
end Const;
```

Utilisation de constante nommée en C++

```
int cons() {
#define C1 ((1 << 100) / 3)    /* L'evaluation aura lieu a l'usage */
#define C2 ((1 << 100) / 9)    /* Et la aussi */
   return 30 * C1 / C2;        /* Ca marche ? */
};
```

Calculs intermédiaires

Malgré les vérifications, Ada autorise des calculs qui sortent temporairement d'un sous-type du moment qu'ils restent dans le type de base. Exemple : prendre 2/3 d'une valeur donnée.

Calcul avec résultat intermédiaire plus grand

```
subtype Small_Integer is Integer range 0 .. 255;

function Scale (I : Small_Integer) return Small_Integer is
begin
   return I * 2 / 3;
end Scale;
```

Ici, le résultat intermédiaire peut ne pas tenir dans l'intervalle défini par **Small_Integer** mais reste dans celui de **Integer**.
C'est une des raisons pour lesquelles on aura intérêt à spécifier les types le plus précisément possible : aucune vérification n'est nécessaire.

Autres vérifications statiques

En disposant des bornes, le compilateur est capable de prédire que certaines opérations vont échouer.

Échec statique

```
type My_Integer is Integer range 10 .. 13;

function Add (X, Y : My_Integer) return My_Integer is
begin
   return X + Y;
end Add;
```

Effet de la compilation

```
function add (x : types__my_integer; y : types__my_integer) return
  types__my_integer is
begin
   return [constraint_error "range check failed"];
end add;

add.adb:4:13: warning: value not in range of type "My_Integer" defined at types.ads:23
add.adb:4:13: warning: "Constraint_Error" will be raised at run time
```

Types de taille fixée

Il est possible de fixer la taille d'un type, et de l'utiliser dans un tableau sans espace perdu.

Tableau « packé »

```
type My_Integer is new Integer range 0 .. 3;
for My_Integer'Size use 2;

type My_Array is array (Natural range <>) of My_Integer;
pragma Pack (My_Array);
```

L'accès aux éléments se fera avec des masques et des décalages de manière transparente pour les utilisateurs.

Accès à des données « packées »

Si les types sont bien spécifiés, l'accès aux éléments ne nécessite pas de vérification et est efficace :

Accès sans besoin de vérification

```
subtype Sub_Index is Natural range 1 .. 16;
subtype Sub_Array is My_Array (Sub_Index)

function Get (A : Sub_Array; I : Sub_Index) return My_Integer
is
begin
   return A (I);
end Get;
```

se traduira par

Version expansée

```
function get (s : types__sub_array; n : types__sub_index) return
  types__my_integer is
begin
   return types__my_integer!(shift_right!(s, 2 * (integer(n - 1))) and
     16#3#);
end get
```

Accès à des données « packées » (suite)

Le code assembleur généré traduit cette efficacité :

Accès sans besoin de vérification

```
_ada_get:
        leal    -2(%rsi,%rsi), %ecx
        shrl    %cl, %edi
        movl    %edi, %eax
        andl    $3, %eax
        ret
```

Origine avec biais

Ada permet également de déclarer des types biaisés. Les calculs sont également optimisés et convertis uniquement lorsque c'est nécessaire.

Type biaisé

```
type My_Bias is new Integer range 10 .. 13;
for My_Bias'Size use 2;

procedure Complement (X : in out My_Bias) is
begin
   X := My_Bias'First + My_Bias'Last - X;
end Complement;
```

Compilation du fragment de code

```
_ada_complement:
        movzbl  %dil, %edi
        movl    $3, %eax
        subl    %edi, %eax
        ret
```

L'opération est effectuée directement sur la représentation biaisée, sans risque de se retrouver avec une valeur invalide.

Inversion d'un tableau de booléens

Dans certains cas, le compilateur peut générer des vérifications réellement non coûteuses. Quel code sera généré pour la procédure `Invert` ?

Inversion douteuse de booléens

```
subtype Always_True is Boolean range True .. True;
type True_Array is array (Natural range <>) of Always_True;

procedure Invert (A : in out True_Array) is
begin
   A := not A;
end Invert;
```

Inversion d'un tableau de booléens (suite)

Le compilateur réalise que le seul cas où l'appel ne doit pas générer d'erreur est lorsque le tableau est vide.

Inversion douteuse de booléens (expansé)

```
procedure bools__invert (a : in out bools__true_array) is
begin
   [constraint_error when
     a'length /= 0
     "range check failed"]
   $system__boolean_array_operations__vector_not (a'address,
     a'address, a'length);
   return;
end bools__invert;
```

Mais faut-il avoir confiance ?

Dans certains cas, il est nécessaire de vérifier que la valeur d'une variable se trouve bien dans la plage de valeurs autorisées :

- lorsque la valeur vient de l'extérieur (saisie par l'utilisateur, fichier, réseau, importation) ;
- lorsque la valeur provient d'une conversion non vérifiable (**Unchecked_Conversion**).

L'attribut **'Valid** permet de s'assurer qu'une variable est valide vis-à-vis de ses contraintes. Son utilisation n'est pas considérée comme une lecture de la valeur, qui risquerait d'engendrer une exception.
Sans cet attribut, les tests de validité effectués avec des opérateurs de comparaison échoueraient, car optimisés comme tout le temps vrais.

Mais faut-il avoir confiance ? (suite)

Et si on oublie d'initialiser une variable ?

Programme incorrect mais légal

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Test is
   subtype N is Natural range 10 .. 13;
   X : N;
   Y : N;
begin
   Put_Line (X'Img);
   Y := X + 10;
   Put_Line (Y'Img);
end Test;
```

Le compilateur prévient qu'il y a un problème, et notamment indique qu'il fera échouer l'addition :

```
test.adb:4:04: warning: variable "X" is read but never assigned
test.adb:8:11: warning: value not in range of type "N" defined at line 3
test.adb:8:11: warning: "Constraint_Error" will be raised at run time
```

L'exécution donne :

```
 0
raised CONSTRAINT_ERROR : test.adb:8 range check failed
```

Mais que peut-on faire ?

Des options de compilation permettent d'ajouter des tests supplémentaires, au détriment des performances :

Code expansé

```
procedure test is
   [...]
   x : n;
begin
   [constraint_error when
     not (interfaces__unsigned_32!(x) >= 10 and then
       interfaces__unsigned_32!(x) <= 13)
     "invalid data"]
[...]
```

On peut choisir où placer les tests : lors des affectations, des appels de sous-programmes, à leur retour, lors de l'utilisation comme indice, etc.

pragma Normalize_Scalars

La directive de compilation **Normalize_Scalars** permet d'affecter des valeurs invalides aux variables non-initialisées lorsque c'est possible.
Par exemple, avec GCC :

- Un **Natural** sera initialisé à -1.
- Un **Positive** sera initialisé à 0.

Seuls les types disposant d'une valeur non-valide représentable dans la taille donnée bénéficieront de cette protection. On peut alors utiliser les vérifications mentionnées ci-dessus pour vérifier leur validité.

pragma Normalize_Scalars (suite)

Dans la plupart des cas, il est possible d'honorer cette directive de compilation de manière efficace.

Exemple d'implémentation

```
procedure N (T : in Boolean; X : out Natural) is
begin
   --  Pas de warning ici, car peut-etre legitime.
   if T then
      X := 42;
   end if;
end N;


procedure n (t : in boolean; x : out natural) is
begin
   x := integer!(16#FFFF_FFFF#);
   if t then
      x := 42;
   end if;
   return;
end n;


_ada_n:
        cmpb    $1, %dil
        sbbl    %eax, %eax
        orl     $42, %eax
        ret
```

Conclusion

- Ada permet de programmer de manière sûre...
- ...tout en restant efficace.
- La plupart des vérifications sont peu coûteuses en pratique.
- Certaines pourraient être plus efficaces (trappes sur les opérations entières).
- Attention : le compilateur a le droit de changer la sémantique du moment qu'une valeur invalide ne se propage pas.
- Les vérifications peuvent être totalement supprimées, ou augmentées (ceinture et bretelles).