Comment migrer du Pascal à l'Ada

1) Introduction

Après avoir eu son heure de gloire sur Macintosh, le langage Pascal est sorti par la petite porte sans faire grand bruit. Tandis que sur PC, Borland popularisait le Pascal, Apple le positionnait au coeur du premier système Macintosh. Il en reste quelques traces dans Carbon - API issue du relookage de la boite à outils du système du premier Mac (Toolbox). Les appels et les paramètres des routines de Carbon conservent certaines structures du Pascal.

Mais, aujourd'hui, les langages C et C++ ont supplantés le Pascal dans cette fonction d'interface avec le système. Bien que très différent du Pascal dans son approche de la programmation, Ada reprend la majeure partie des structures du Pascal. Il devient ainsi assez facile de transposer ou de traduire un programme source Pascal en Ada. Une grande aide nous vient de l'utilitaire "P2Ada" qui opère automatiquement la majeure partie de cette traduction.

Il manque deux composantes pour poursuivre une fois la traduction effectuée. La première est bien entendu un compilateur Ada. Aujourd'hui le compilateur GNAT en distribution libre est présent sur la plupart des plates-formes, notamment gnat-osx est disponible sur Mac OS X. Il est particulièrement intégré dans le dernier système du Macintosh, puisqu'il est présent dans XCode - l'environnement de programmation natif du système.
La seconde composante est l'interface de programmation (API) avec le système. Celle ci est pourvu sur Mac OS X à travers l'accès en Ada à l'API Carbon.

L'intérêt de cette solution est de proposer simplement une seconde vie sur un système moderne à des programmes laissés à l'abandon sans pour autant repartir de zéro. Les modifications manuelles sont principalement limitées aux conversion de types des nombres entiers (Ada étant moins laxiste que Pascal dans ce domaine), la prise en compte du type "string" particulier au Pascal et les spécificités liées au matériel.

Nous pouvons ainsi avec Mac OS X et Ada bénéficier des avantages d'un système et d'un langage sûrs et modernes.

2) Installation de P2Ada

Récupérez sur le bureau du Mac l'archive newp2ada.zip à l'adresse
http://homepage.sunrise.ch/mysunrise/gdm/gsoft_fr.htm#p2ada
(La version utilisée est celle du 15 novembre 2006, voir les évolutions dans le fichier whatsnew.txt)

```
$ cd ~/Desktop
$ mkdir newp2ada
$ cd newp2ada
$ unzip -aaLL ~/Desktop/newp2ada.zip
```

Modifier le fichier p2adopti.ads pour avoir des fins de ligne de type Unix et un modulo conforme à la définition du Think Pascal puis lancer la compilation :

```
$ cd newp2ada
$ vi p2adopti.ads
/DOS<CR>
n
cwUnix<esc>
  new_line_endings : constant Line_endings:= Unix;
/mod<CR>
n
cwrem<ESC>
  translation_of_MOD: constant Mod_Rem:= is_rem;
ZZ

$ make
```

3) Utilisation de P2Ada

P2Ada s'utilise simplement comme ceci :

```
$ ./p2ada < nptest0.pas > nptest0.adb
$ gnatmake nptest0
$ ./nptest0
```

Les directives de compilation sont traitées par BP2P :

```
$ ./bp2p nptest1.pas -dcoucou -\$z+ | ./p2ada > nptest1.adb
```

4) Traduction d'un programme Pascal simple

Prenons par exemple le programme d'attribution simple présenté la première fois sur le site avec une implémentation Pascal. Le programme est simple il ne fait appel qu'à des entrées / sorties textes.
Nous employons directement P2Ada et gnatmake à la suite.

```
$ p2ada < attrib.p > attrib.adb
$ gnatmake -g -gnatf attrib
gcc -c -g -gnatf attrib.adb
attrib.adb:61:28: "TlisteAttrib" is undefined
attrib.adb:63:53: "TlisteAttrib" is undefined
attrib.adb:63:53: instantiation abandoned
attrib.adb:96:18: expected type "PListeAttrib" defined at line 61
attrib.adb:96:18: found type access to "Tlisteattrib" defined at line 96
attrib.adb:149:07: "Dispose" is undefined
attrib.adb:162:13: "Dispose" is undefined
attrib.adb:169:13: "Dispose" is undefined
attrib.adb:230:07: assignment to "in" mode parameter not allowed
attrib.adb:232:09: assignment to "in" mode parameter not allowed
attrib.adb:280:29: invalid operand types for operator "And"
attrib.adb:280:29: left operand has type "Standard.Integer"
attrib.adb:280:29: right operand has a composite type
attrib.adb:286:26: invalid operand types for operator "And"
attrib.adb:286:26: left operand has type "Standard.Integer"
attrib.adb:286:26: right operand has a composite type
attrib.adb:299:50: invalid operand types for operator "And"
attrib.adb:299:50: left operand has type "Standard.Integer"
attrib.adb:299:50: right operand has a composite type
attrib.adb:334:01: no candidate interpretations match the actuals:
attrib.adb:334:01: too many arguments in call to "Put"
attrib.adb:334:16: expected private type "Ada.Text_Io.File_Type"
attrib.adb:334:16: found type "Standard.Integer"
attrib.adb:334:16:   ==> in call to "Put" at a-tienio.ads:56, instance at line 48
attrib.adb:334:16:   ==> in call to "Put" at a-tiflio.ads:80, instance at a-lfteio.ads:18
attrib.adb:334:16:   ==> in call to "Put" at a-tiflio.ads:69, instance at a-lfteio.ads:18
attrib.adb:334:16:   ==> in call to "Put" at a-tiflio.ads:62, instance at a-lfteio.ads:18
attrib.adb:334:16:   ==> in call to "Put" at a-tiflio.ads:80, instance at a-flteio.ads:20
attrib.adb:334:16:   ==> in call to "Put" at a-tiflio.ads:69, instance at a-flteio.ads:20
attrib.adb:334:16:   ==> in call to "Put" at a-tiflio.ads:62, instance at a-flteio.ads:20
attrib.adb:334:16:   ==> in call to "Put" at a-tiinio.ads:61, instance at a-inteio.ads:18
gnatmake: "attrib.adb" compilation error
```

Un premier type d'erreur apparaît avec l'instruction 'new' utilisée avec le type pointeur en Pascal et le type source en Ada. La modification est d'ordinaire simple, il suffit de mettre une déclaration incomplète du type source avant sont utilisation avec le pointeur comme l'indique obligeamment le commentaire de P2Ada. Il faut également déplacer la procédure "Dispose" après la déclaration complète du type source.

```
> type TlisteAttrib;

< procedure Dispose is new Ada.Unchecked_Deallocation(TlisteAttrib,
PListeAttrib);
```

Un second type d'erreur est lié au fait que Ada n'autorise pas l'utilisation d'un paramètre comme variable locale. La solution consiste à ajouter une variable locale initialisée avec le paramètre.

```
<               NivChoix : Integer;
>               ArgNivChoix : Integer;

>       NivChoix : Integer := ArgNivChoix;
```

Un troisième type d'erreur est lié au fait que Ada ne supporte pas directement les types ensembles du Pascal. La correction est moins évidente, ici le test des bornes minimum et maximum suffit car l'ensemble est celui des nombres entiers.

```
<       exit when NbClientes and (1 .. CMaxClientes => True, others => False)  --
[P2Ada]: "x in y" -> "x and y" redefine "and" before
>       exit when (NbClientes >= 1) or (NbClientes <= CMaxClientes)  -- [P2Ada]: "x
in y" -> "x and y" redefine "and" before

<       exit when NbRobes and (1 .. CMaxElements => True, others => False)  --
[P2Ada]: "x in y" -> "x and y" redefine "and" before
>       exit when (NbRobes >= 1) or (NbRobes <= CMaxElements)  -- [P2Ada]: "x in
y" -> "x and y" redefine "and" before

<           exit when TabClientes(Cliente, Choix) and (1 .. CMaxElements => True,
others => False)  -- [P2Ada]: "x in y" -> "x and y" redefine "and" before
>           exit when (TabClientes(Cliente, Choix) >= 1) or (TabClientes(Cliente,
Choix) <= CMaxElements)  -- [P2Ada]: "x in y" -> "x and y" redefine "and" before
```

Dans le cas où l'ensemble est discret, par exemple un ensemble d'énumérés, la solution consiste à redéfinir un type tableau ensemble et une surcharge de l'opérateur "and" avec ce type comme ceci dans le cas de l'ensemble des caractères :

```
type TChar is array (Character) of Boolean;
function "and" (Item : Character; Of_Set : TChar) return Boolean is
begin
return Of_Set(Item);
end;
```

Un quatrième type d'erreur est plus classique car lié au renforcement du typage d'Ada par rapport à Pascal. Nous devons signifier explicitement à Ada l'affichage d'un type réel.

```
< Put(Performance/NbClientes,4,1,0); New_Line;
> Put(Float(Performance/NbClientes),4,1,0); New_Line;
```

On relance.

```
$ gnatmake -g -gnatf attrib
gcc -c -g -gnatf attrib.adb
attrib.adb:80:08: warning: variable "NivChoix" is never read and never assigned
attrib.adb:80:17: warning: variable "Element" is never read and never assigned
attrib.adb:82:07: warning: declaration hides "NivChoix" at line 80
attrib.adb:82:07: warning: for loop implicitly declares loop variable
attrib.adb:83:09: warning: declaration hides "Element" at line 80
attrib.adb:83:09: warning: for loop implicitly declares loop variable
attrib.adb:107:08: warning: variable "NivChoix" is never read and never assigned
attrib.adb:107:17: warning: variable "Element" is never read and never assigned
attrib.adb:111:07: warning: declaration hides "NivChoix" at line 107
attrib.adb:111:07: warning: for loop implicitly declares loop variable
attrib.adb:112:09: warning: declaration hides "Element" at line 107
attrib.adb:112:09: warning: for loop implicitly declares loop variable
attrib.adb:140:07: warning: variable "NivChoix" is never read and never assigned
attrib.adb:152:07: warning: declaration hides "NivChoix" at line 140
attrib.adb:152:07: warning: for loop implicitly declares loop variable
attrib.adb:196:07: warning: variable "Iterration" is never read and never assigned
attrib.adb:200:07: warning: declaration hides "Iterration" at line 196
attrib.adb:200:07: warning: for loop implicitly declares loop variable
attrib.adb:224:08: warning: variable "Element" is never read and never assigned
attrib.adb:228:07: warning: declaration hides "Element" at line 224
attrib.adb:228:07: warning: for loop implicitly declares loop variable
attrib.adb:242:09: warning: variable "Cliente" is never read and never assigned
attrib.adb:247:09: warning: declaration hides "Cliente" at line 242
attrib.adb:247:09: warning: for loop implicitly declares loop variable
attrib.adb:275:26: warning: variable "Choix" is never read and never assigned
attrib.adb:275:32: warning: variable "Cliente" is never read and never assigned
```

```
attrib.adb:297:09: warning: declaration hides "Choix" at line 275
attrib.adb:297:09: warning: for loop implicitly declares loop variable
attrib.adb:308:07: warning: declaration hides "Cliente" at line 275
attrib.adb:308:07: warning: for loop implicitly declares loop variable
gnatbind -x attrib.ali
gnatlink attrib.ali -g
```

Le compilateur nous indique des variables non utilisée. Il s'agit de variables de boucle, implicites en Ada. Nous les éliminons. Ce n'est pas obligatoire puisque la variable de boucle surcharge la variable locale mais c'est plus propre.
Nous pouvons lancer le programme.

```
$ ./attrib
Attribution "performante".
Entrer le nombre de clientes (1 .. 10) : 2
Entrer le nombre de robes (1 .. 10) : 2
Choix de la cliente A :
Entrer la robe choisie (1 .. 10) : 1
Entrer la robe choisie (1 .. 10) : 2
Choix de la cliente B :
Entrer la robe choisie (1 .. 10) : 2
Entrer la robe choisie (1 .. 10) : 1

Attribution de la robe          1 a la cliente A (Choix          1).
Attribution de la robe          2 a la cliente B (Choix          1).

Performance :   1.0
```

4) Traduction d'un programme Pascal graphique

Avec XCode, nous allons créer un projet du type "Ada Carbon Application" nommé "lanceur" (voir l'utilisation de XCode et Carbon avec GNAT sur Blady). Nous allons traduire un programme affichant une courbe fractale de Julia. Le programme en Think Pascal a été écrit par Thomas & Jean-Edouard Lachand-Robert auteurs du très précieux "Grand Livre de la Programmtion Mac".

```
$ p2ada < julia.p > julia.adb
```

Nous ajoutons le fichier "julia.adb" dans le projet.
Nous créons dans XCode un nouveau fichier "julia.ads" avec juste ceci :

```
procedure Julia;
```

Nous modifions le fichier "lanceur.adb" en ajoutant dans l'entête :

```
with Julia;
```

Nous remplaçons l'appel "DoAboutBox" par "Julia".

Sur le fichier "julia.adb" nous effectuons plusieurs modifications :

a) Nous éliminons toutes les unités importées par "p2ada".

```
--with Ada.Text_IO;                          use Ada.Text_IO;
--with Ada.Integer_Text_IO;                  use Ada.Integer_Text_IO;
--with Ada.Float_Text_IO;                    use Ada.Float_Text_IO;
--with Ada.Long_Float_Text_IO;               use Ada.Long_Float_Text_IO;
--with Ada.Direct_IO;
--with Ada.Command_Line;                     use Ada.Command_Line; --
ParamStr,...
--with Ada.Characters.Handling;              use Ada.Characters.Handling; --
UpCase
--with Interfaces;                           use Interfaces; -- For
Shift_Left/Right
-- This is for Pi :
--with Ada.Numerics;                         use Ada.Numerics;
-- This is for Sqrt, Sin, Cos, etc. :
--with Ada.Numerics.Elementary_Functions; use
Ada.Numerics.Elementary_Functions;
--with Ada.Numerics.Long_Elementary_Functions;
-- use Ada.Numerics.Long_Elementary_Functions;
-- This is for Dispose. P2Ada writes automatically:
--   "Dispose is new Ada.Unchecked_Deallocation(<type>, <pointer type>)".
--with Ada.Unchecked_Deallocation;
```

et

```
--  package Boolean_Text_IO is new Enumeration_IO(Boolean);
--  use Boolean_Text_IO; -- [P2Ada]: This is for 'Write([Boolean])'
--  package Byte_Direct_IO is new Ada.Direct_IO(Unsigned_8);
 -- [P2Ada]: This is for 'file' without type
--  Program_halted: exception; -- [P2Ada]: This is for the Halt pseudo-
procedure
```

b) Nous ajoutons les unités de Carbon Ada :

```
with ApplicationServices.QD.Quickdraw;
use ApplicationServices.QD.Quickdraw;
with Carbon.HIToolBox.Events; use Carbon.HIToolBox.Events;
```

c) Nous ajoutons les unités de Think Pascal (voir sur Blady) dans le source et dans le projet :

```
with thp.system; use thp.system;
with thp.windows; use thp.windows;
```

d) Les nombres entiers et réels doivent être explicitement définis :

```
            P2Ada_Var_1.y := Sqrt((P2Ada_Var_1.x - a) / 2.0);
            P2Ada_Var_1.x := Sqrt((P2Ada_Var_1.x + a) / 2.0);
            if  b < 0.0 then
                i := Integer(Random)  rem  2;
            P2Ada_Var_1.x := Float(i) * P2Ada_Var_1.x;
```

```
            P2Ada_Var_1.y := Float(i) * P2Ada_Var_1.y;
        pt.x := 1.0;
        pt.y := 1.0;
MoveTo(100 + Short_Integer(Round(pt.x * 80.0)), 100 + Short_Integer(Round
(pt.y * 80.0)));
```

e) Nous ajoutons l'appel à "SystemTask" permettant de redonner la main au système :
SystemTask;

f) Dans la cible XCode (Target) "lanceur" ajoutons la variable "MAIN_PROGRAM=lanceur.adb" dans les "Build Settings" pour que XCode ne prenne pas la procédure "Julia" comme programme principal.

Nous lançons la compilation du projet et l'exécution du programme puis nous sélectionnons "About..." dans le menu de l'application.

5) Traduction d'un programme Pascal Objet

Avec XCode, nous allons créer un projet du type "Ada Carbon Application" nommé "lanceur" (voir l'utilisation de XCode et Carbon avec GNAT sur Blady). Nous allons traduire un programme affichant des objets graphiques. Le programme en Think Pascal est un exemple venant avec ce compilateur.

Think Pascal permet de ne pas réécrire les paramètres des procedures de la partie implémentation d'une unité, au contraire de P2Ada qui attend ces paramètres. Je les ais ajoutés dans ObjIntf.p.
La traduction utilise alors la spécificité de P2Ada de pouvoir enregistrer dans un fichier les éléments déclarés dans une unité et de les réutilisés ensuite.

```
$ p2ada ObjIntf.p -eobjintf.def > objintf.adb
$ gnatchop -w objintf.adb
$ p2ada UShapes.p -eushapes.def -iobjintf.def > ushapes.adb
$ gnatchop -w ushapes.adb
$ p2ada LearnOOP.p -iushapes.def > learnoop.adb
```

Nous créons un nouveau fichier "learnoop.ads" avec juste ceci :

```
procedure Learn00P;
```

Nous modifions le fichier "lanceur.adb" en ajoutant dans l'entête :

```
with Learn00P;
```

Nous remplaçons l'appel "DoAboutBox" par "LearnOOP".

Think Pascal considère automatiquement toute déclaration d'objet comme étant un pointeur sur l'objet proprement dit. Pour Ada nous devrons donc explicitement déclarer ce pointeur.

Sur le fichier "objintf.ads" nous effectuons plusieurs modifications :

a) Déclarons les types pointeurs sur l'objet de base :

```
type
    BTObject  is
       tagged record  null;end record;
   type TObject is access all BTObject'class;
```

b) Les méthodes deviennent :

```
function  ShallowClone(Self:TObject) return  TObject;
function  Clone(Self:TObject) return  TObject;
procedure ShallowFree(Self: in out TObject);
procedure Free(Self: in out TObject);
```

Sur le fichier "objintf.adb" nous effectuons plusieurs modifications :

a) Ajoutons les clauses with :

```
with CoreServices.CarbonCore.MacTypes; use
CoreServices.CarbonCore.MacTypes;
with CoreServices.CarbonCore.MacMemory; use
CoreServices.CarbonCore.MacMemory;
```

b) Ajoutons les fonctions de conversion de pointeur :

```
   function Convert is
        new Unchecked_Conversion(TObject, Handle);
    function Convert is
        new Unchecked_Conversion(Handle, TObject);
```

c) Modifions l'utilisation des objets et des pointeurs :

```
  function ShallowClone(Self:TObject) return  TObject --  modif PP
              result: aliased Handle;
    begin
          result := Convert(SELF);
if  HandToHand(result'access) /= noErr then
...
        Result_ShallowClone := Convert(result);
```

Sur le fichier "shapes.ads" nous effectuons plusieurs modifications :

a) Ajoutons la clause use duale :

```
  use
        ObjIntf;
```

b) Ajoutons les clauses with :

```
with CoreServices.CarbonCore.MacTypes; use
CoreServices.CarbonCore.MacTypes;
```

c) Déclarons les types pointeurs sur l'objet de base :

```
  type
  BCShape  is  new BTObject with record
                itsRectangle: aliased Rect;
end record;
   type CShape is access all BCShape'class;
```

d) Les méthodes deviennent :

```
  procedure  SetBounds(Self: in out CShape; aTop, aLeft, aBottom, aRight:
integer);
  procedure Draw(Self: in out CShape);
```

Sur le fichier "shapes.adb" nous effectuons plusieurs modifications :

a) Ajoutons les clauses with :

```
  with ApplicationServices.QD.Quickdraw; use
ApplicationServices.QD.Quickdraw;
```

b) Modifions l'utilisation des objets :

```
  procedure SetBounds (Self: in out CShape; aTop, aLeft, aBottom, aRight:
integer)
```

c) Modifions l'utilisation implicite du with Pascal :

```
                P2Ada_Var_1.top := Short_Integer(aTop);
                P2Ada_Var_1.left := Short_Integer(aLeft);
                P2Ada_Var_1.bottom := Short_Integer(aBottom);
                P2Ada_Var_1.right := Short_Integer(aRight)
```

Sur le fichier "learnoop.adb" nous effectuons plusieurs modifications :

a) Remontons la liste des unités "with" avec les autres comme indiqué dans le commentaire et ajoutons la clause use duale :

```
    with
        ObjIntf, Shapes; -- [P2Ada]: place it before main procedure
    use
        ObjIntf, Shapes;
```

b) Ajoutons les clauses with pour les procédures graphiques :

```
with thp.windows; use thp.windows;
```

c) Contournons la contrainte des paramètres d'une fonction par un pointeur :

```
         aShape:aliased CShape;
    function  ChooseShape( theShape: access  CShape) return  Boolean
                theShape.all:= new BCRectangle;
```

```
        while ChooseShape (aShape'access) loop
```

d) Ajoutons les procédures pour l'affichage :

```
ShowDrawing;
...
SystemTask;
```

Nous lançons la compilation du projet avec la bibliothèque Carbon et l'exécution du programme puis nous sélectionnons "About..." dans le menu de l'application.

$ gnatmake learnoop -largs /System/Library/Frameworks/Carbon.framework/Carbon

Certains de ces conseils de traduction Pascal vers Ada se trouvent également sur le site oueb de Gautier de Montmollin :
http://homepage.sunrise.ch/mysunrise/gdm/pascada.htm

Pascal Pignard, octobre, décembre 2002, janvier, avril 2003, novembre-décembre 2006, février 2007.