# Correction du Partiel TYPO & CMP

EPITA – Promo 2007 – **Tous documents autorisés** *

Mai 2005 (1h30)

Le sujet et une partie de sa correction ont été écrits par Akim Demaille.

Une copie synthétique, bien orthographiée, avec un affichage clair des résultats, sera toujours mieux notée qu'une autre demandant une quelconque forme d'effort de la part du correcteur. Une argumentation informelle mais convaincante, sera souvent suffisante.

## 1 Typologie des Langages

1. Pourquoi les pointeurs sont-ils dangereux en C ?

   **Correction:** Un pointeur C est une adresse mémoire qui ne correspond par forcément à un objet (l'exemple canonique est NULL, qui désigne une adresse mémoire interdite) ; rien ne garantit donc qu'une valeur manipulée par pointeur soit valide.

2. Pourquoi les références en C++ sont-elles des pointeurs policés ?

   **Correction:** Une référence en C++ est également une adresse mémoire, mais qui doit être initialisée via un objet existant ; on est assuré qu'une valeur manipulée par référence est valide (tant que l'objet référencé l'est, bien entendu). Il s'agit donc d'un mécanisme plus sûr qu'un pointeur.

3. Pourquoi les unions sont-elles dangereuses en C ?

   **Correction:** On rappelle qu'une union est un *type somme*, construit comme la réunion disjointe de plusieurs types. Comme une structure, une union définit des membres, mais la valeur d'au plus un membre peut-être stockée à un instant donné. La taille d'une union est égale à celle de son membre le plus large.
   Les unions du C introduisent des faiblesses dans le typage, en ce sens que le langage n'offre aucun mécanisme pour connaître le membre réellement contenu dans l'union, ni à la compilation, ni à l'exécution. C'est donc à l'utilisateur de préciser le membre qu'il veut utiliser, ce qui laisse la porte ouverte aux erreurs inhérentes au typage faible.
   Par exemple, le programme suivant

```c
#include <stdio.h>

int main ()
{
  union { int i; char* s; } f;
  f.s = "Hello World!";
  f.i = 0;                      // Warning, overwriting f.s!
  puts (f.s);
  return 0;
}
```

   meurt sur un SEGV.

4. Quelle restrictions sur les membres d'une union le C++ pose-t-il ?

*"Tout document autorisé" signifie que notes de cours, livres, annales, etc. sont explicitement consultables pendant l'épreuve. Le zèle de la part des surveillants n'a pas lieu d'être, mais dans un tel cas contacter le LRDE au 01 53 14 59 22.

**Correction:** En C++, une union ne peut contenir :

(a) des membres ayant un constructeur, un destructeur ou un `operator=` « non triviaux » (définis par l'utilisateur), ou des tableaux de tels membres ;

(b) des attributs déclarés `static` ;

(c) des membres références ;

(d) des méthodes virtuelles (mais elle peut contenir des méthodes non virtuelles).

(La réponse attendue était la première.)

5. Qu'est-ce qui motive cette limitation ?

**Correction:** Rappelons que tous les membres d'une classe (au sens large, ce qui inclut les entités définies avec `class`, `struct` et `union`) doivent être initialisés à la construction d'une instance de cette classe.
Prenons le cas d'une structure (non union) qui comporterait deux membres possédant un constructeur non trivial, comme dans le programme suivant :

```
#include <iostream>

struct A { A () { std::cout << "A::A" << std::endl; } };
struct B { B () { std::cout << "B::B" << std::endl; } };

int main ()
{
  struct
  {
    A a;
    B b;
  } foo;
}
```

Ici, la construction de `foo` initialise ses membres `a` et `b` en utilisant leurs constructeurs sans argument (non triviaux), produisant un affichage sur la sortie standard. Mais imaginez que nous puissions remplacer `struct` par `union` dans ce code ; sachant qu'une union contient au plus l'un de ses membres, que doit faire le compilateur à la construction de `foo` ? Doit-il initialiser `a`, ou `b`, ou les deux, ou aucun ? On constate que l'initialisation d'une union est assez différente de celle d'une classe ou d'une structure, et qu'elle introduit son lot d'ambiguïtés, d'où la limitation évoquée dans la question précédente.

6. Proposer un mécanisme généralisant les unions du C pour fonctionner pour « tout » le C++. On s'intéressera particulièrement à (i) l'installation, (ii) la copie, (iii) la destruction.

**Correction:** Un tel mécanisme est présent dans la bibliothèque Boost, via la classe paramétrée `boost::variant<>` (`http://www.boost.org/doc/html/variant.html`). Celle-ci permet de définir des classes qui s'apparentent à des unions, où le type de chaque membre est un paramètre de `boost::variant<>`.
La construction par défaut crée un `variant` contenant un membre du premier type :

```
boost::variant<int, std::string> v;     // v contains an integer.
```

La classe fournit également des constructeurs pour chaque type de membres. L'affectation d'un variant à un autre variant effectue des affectations ou des constructions/destructions de membres, selon que les types des contenus dans les deux variants sont les mêmes ou non. La destruction d'un variant provoque la destruction de son contenu.

7. À quoi ressembleraient des fonctions unaires qui prennent en argument une telle union généralisée ?

**Correction:** De telles fonctions devraient traiter chaque cas séparément, à la manière du filtrage sur types sommes (variants) de Caml :

```
type foo = Int of int | Str of String.t;;

let print_foo = function
    Int i -> print_int i
  | Str s -> print_string s
;;
```

Les variant de Boost proposent deux solutions :

– une simple, qui consiste à extraire le contenu de l'union en précisant le type attendu : `boost::get<std::string> (v)`, et qui déclenche une erreur à l'exécution en cas de mauvais type ;
– une élégante, où c'est un visiteur qui est utilisé pour effectuer un traitement sur l'union : `boost::apply_visitor (my_visitor (), v)`

## 2 Construction des Compilateurs : Désucrage sucré

1. Qu'est-ce que le sucre syntaxique ?

   **Correction:** Il s'agit de constructions syntaxiques supplémentaires proposées par un langage n'apportant rien de plus au niveau de sa syntaxe abstraite, mais permettant des variantes d'écriture. Le sucre syntaxique apporte souvent simplicité, lisibilité, raccourcis d'écriture, confort, etc. Il peut permettre l'expression de nouveaux concepts (la surcharge de fonctions peut être considérée comme telle, par exemple).

2. Qu'est-ce que le désucrage ?

   **Correction:** Le désucrage est l'opération qui consiste à remplacer les éléments de sucre syntaxique par des constructions de syntaxe de base (« non sucrées »).

3. Citer les 5 phases d'une partie frontale d'une compilateur comme `tc`.

   **Correction:** (i) scanner, (ii) parser et construction de l'AST, (iii) liaison des noms, (iv) typage, (v) traduction vers une représentation intermédiaire.

4. Discuter deux emplacements possibles d'une phase de désucrage, et en comparer les mérites respectifs.

   **Correction:**
   (a) Pendant la construction de l'AST
   (b) entre le parseur et la liaison
   (c) entre la liaison et le typage
   (d) après le typage
   Plus c'est tôt, moins ça a d'impact sur le compilateur, mais moins les messages sont clairs pour l'utilisateur. Plus c'est tard, plus le sucre peut être complexe.

5. On considère l'ajout de + entre chaînes de caractères dans `tc` comme sucre pour la primitive `concat`. On autorise par exemple `print (dir + "/" + file)`.
   Où placer sa phase de désucrage ?

   **Correction:** Il est nécessaire d'avoir les informations de typage pour faire une chose pareille : `a + b` se traduit différemment pour les `string` et les `int`. Il faut donc passer après le type-checking.

6. Proposer une implémentation pour désucrer ce cas. On ne demande pas le code, mais les grandes lignes de la mise en œuvre.

   **Correction:** Il s'agit bien sûr d'un visiteur supplémentaire, qui, lorsqu'il tombe sur une BinOp, dans le cas du + entre `string` produise en remplacement l'AST pour `concat (left, right)` où `left` et `right` sont les versions désucrées des fils.

7. Quelle difficulté introduit l'appel à une fonction ou à une primitive ? On pensera particulièrement à la nécessité pour la suite du compilateur de ne pas avoir à se préoccuper du fait qu'il y ait eu, ou pas eu, de phase de désucrage : il compte sur la liaison des noms.

   **Correction:** L'appel d'une fonction/primitive doit être lié : on veut savoir que le `concat` ci-dessus est la primitive. Comment lier ? On ne peut produire l'appel puis simplement relancer le `BindVisitor`, parce que dans ce cas on peut appeler un mauvais `concat` :

   ```
   let
     function concat (a: int, b: int) : int = a + b
   in
     print ("Oh oh..." + "\n")
   end
   ```

   Ici, on aurait sans doute un magnifique SEGV. Notez que le `TypeVisitor` étant déjà passé, à moins de ne le relancer, l'erreur de type ne sera pas détectée.

8. Pourquoi les macros du C ne sont pas (toujours) une bonne réponse pour introduire du sucre ?

   **Correction:** Les macros du C ne supportent pas la récursion, ce qui peut être gênant pour introduire du sucre syntaxique, sachant qu'un arbre de syntaxe abstraite est essentiellement une structure définie récursivement !
   Par ailleurs, ces macros ne sont que lexicales, ce qui introduit des risques de changement de sens. Considérons par exemple

   ```
   #define max(A, B) A < B ? B : A

   (1 + max (i++, j++) + 1)
   ```

   qui ne fait pas ce qu'on voudrait (priorité d'opérateurs, effets de bords). Un traitement sur l'AST est ici nécessaire.

9. Traduire

   ```
   for i := l to u do b
   ```

   en une boucle `while`, en conservant à l'esprit les points suivants.
   - Les métavariables *i*, *l*, *u*, et *b* représentent du *code*, pas nécessairement des variables Tiger. Elles sont en quelque sorte les AST fils du `ForExp`.
   - Si $l > u$ le corps de la boucle n'est pas évalué.
   - Si *l* ou *u* vaut `MAX_INT`, le comportement doit être correct et ne doit pas boucler indéfiniment sous prétexte que `MAX_INT + 1 = MIN_INT`.
   - Introduire de nouveaux noms comporte le risque de masquer ceux de l'utilisateur.
   - 9 vient après 8.
   - Le correcteur aime voir *une* réponse, pas une succession de tentatives rayées, blancotées, stabilotées, ciseautées, agrafées, tronçonneuseotées.

   **Correction:**

   ```
   let
     var _l := l'
     var _u := u'
     var i := _l
   in
     if i <= _u then
       while 1 do
         (
           b';
           if i = _u then
             break;
           i := i + 1
         )
   end
   ```

**Correction:** On notera que :

– $l$, $u'$ et $b$ peuvent eux-mêmes inclure des boucles, ce pourquoi on utilise ci-dessus $l'$, $u'$ et $b'$ qui sont leurs versions *désucrées*.

– Comme on a besoin de $u$ plusieurs fois, il faut passer par une variable temporaire pour n'évaluer $u$ qu'une seule fois.

– Par conséquent il faut aussi stocker $l$ dans une variable intermédiaire car

```
let
  var _u := u
  var i := l
in
```

évalue $u$ avant $l$.

– Les variables temporaires utilisent des noms interdit à l'utilisateur pour éviter qu'elles ne masquent celles de l'utilisateur. Par exemple transformer

```
let
  var lo = 0
in
  for i := lo + 1 to lo + 1 do print_int (lo + i)
end
```

en

```
let
  var lo := 0
in
  let
    var lo := lo + 1
    var up := lo + 1
    var i := lo
  in
    if i <= up then
      (
        print_int (lo + i);
        while (i < up) do
          (
            i := i + 1;
            print_int (lo + i)
          )
      )
  end
end
```

est très faux : plusieurs `lo` se mélangent.

– $b'$ doit forcément être inclu dans le corps de `while`, dans l'éventualité où il contiendrait une instruction `break` (qui ne peut apparaître que dans une boucle). Ce pourquoi la solution ci-dessous est incorrecte :

```
let
  var _l := l'
  var _u := u'
  var i := _l
in
  if i <= _u then
    (
      b';
      while (i < _u) do
        (
          i := i + 1;
          b'
        )
    )
end
```

10. Pour votre information, notre première implémentation de cette transformation comprend 104 lignes de pénible instantiation à la main de nœuds de la hiérarchie `ast::Ast`. Elle est donc bien sûr difficile à lire, donc à contrôler. Que proposez-vous pour faciliter cette transformation ? On demande ici les grands principes d'une implémentation tenant compte de tous les problèmes soulevés dans la question 9.

**Correction:** Une idée serait d'utiliser directement la syntaxe concrète du langage, plutôt que construire les nœuds de la syntaxe abstraite « à la main ». Le parser serait ensuite relancé sur le code désucré pour obtenir le nouvel AST.
Grosso modo, dans le cas du désucrage des boucles `for`, on souhaite remplacer un sous-arbre de l'AST par un autre sous-arbre, tout en conservant les fils du premier sous-arbre (une copie désucrée, en fait) pour les attacher au deuxième sous-arbre. Le désucrage consiste donc à relancer le parser sur un bout de syntaxe abstraite comportant des « trous » ; ils s'agit du code donné dans la réponse à la question 9 :

```
let
  var _l := l' /* « hole » */
  var _u := u' /* « hole » */
  var i /* « hole » */ := _l
in
  if i /* « hole » */ <= _u then
    while 1 do
      (
        b';
        if i /* « hole » */ = _u then
          break;
        i /* « hole » */ := i /* « hole » */ + 1
      )
end
```

Les trous sont les morceaux de code désignés par les métavariables $i$, $l'$, $u'$, $b'$. En traduisant les AST de ces dernières en syntaxe concrète (via le *pretty printer*), on peut boucher les trous du code, et obtenir un programme complet (et désucré) en syntaxe concrète.

11. Critiquer la solution que vous proposez à la question 10.

**Correction:** La solution précédente présente l'inconvénient de devoir relancer le parser (et de construire et détruire des AST) à chaque désucrage de boucle `for`, ce qui est loin d'être performant !