Correction du Partiel de Compilation

Promo 2004 - Septembre 2002

Une rédaction simple mais convaincante et une mise en évidence des résultats seront très appréciées des correcteurs...

Le taux de succès au premier exercice (considéré comme trivial pour la fin d'une première année du cycle d'ingénieur EPITA) coefficiente la note totale : $N = n_1 \times (n_2 + n_3)$.

1 Contrôle de Connaissances

1. Qu'affiche le programme suivant :

```
#include <iostream>
struct Foo {
  virtual void foo () const { std::cout << "Foo::foo()" << std::endl; }
          void bar () const { std::cout << "Foo::bar()" << std::endl; }
};
struct Bar : public Foo {
  virtual void foo () const { std::cout << "Bar::foo()" << std::endl; }
          void bar () const { std::cout << "Bar::bar()" << std::endl; }
};
int main () {
  const Foo &o1 = Foo (); o1.foo (); o1.bar ();
  const Foo &o2 = Bar (); o2.foo (); o2.bar ();
  const Bar &o3 = Bar (); o3.foo (); o3.bar ();
}
```

Correction:

```
Foo::foo()
Foo::bar()
Bar::foo()
Foo::bar()
Bar::foo()
Bar::bar()
```

2. Écrire un *class template* `Pair` qui stocke deux éléments, `a` et `b`, d'un même type, avec un constructeur prenant les deux valeurs.

Correction: Commencez par remarquer que j'ai écrit "**un** class template", et non pas "une" : c'est normal, puisqu'ici le nom est "template" et l'adjectif "class". Puisque l'on dit "un template", on dit "un class template" de la même façon qu'on dit "un patron de classe" et non une "patron de classe".
Par exemple :

```
template < typename T >
class Pair
{
public:
  Pair (const T &a, const T &b): _a (a), _b (b) {}
private:
  T _a, _b;
};
```

Erreurs/différences acceptées sans perte :

(a) Absence de ; à la fin.

(b) Pas de qualificatif ou bien `class` au lieu de `typename`.

(c) Pas d'utilisation des références.

(d) Oubli du `const` dans le cas de référence.

(e) Pas de `private`.

Erreurs avec perte de 50% de la note :

(a) Oubli du `public` ou équivalent.

3. Utiliser la classe précédente pour déclarer une variable `pi` qui contienne la paire $(3, 14159)$.

Correction:

```
Pair <int> pi (3, 14159);
```

4. Qu'est-ce qu'un `const_iterator` dans STL ?

Correction: Un itérateur en lecture seulement, i.e., un itérateur qui pointe vers les éléments du conteneur en lecture seulement. Si `i` est un `const_iterator`, alors `*i` est `const`.

5. Écrire une fonction `sum` qui prenne une liste STL d'`int` par référence, et en retourne la somme. On prendra garde à la constance et au `namespace`.

Correction:

```
int
sum (const std::list <int> &l)
{
  int res = 0;
  std::list <int>::const_iterator i;
  for (i = l.begin (); i != l.end (); ++i)
    res += *i;
  return res;
}
```

2 Calcul des Liens Statiques

En Tiger, toutes les fonctions n'ont pas besoin d'un lien statique ("Static Link"), comme par exemple l'implémentation traditionnelle de `fact`.

1. Le lien statique sert à deux choses différentes en Tiger. Dans le cas le plus évident, il sert à retrouver le frame dans lequel est logée une variable non locale. Quelle est l'autre cas où l'on utilise le lien statique ?

   Correction: On calcule également le lien statique au moment où l'on appelle une fonction, puisque c'est l'appelant qui sait quelle est la valeur du lien statique à passer à l'appelé.

2. Caractériser précisément les fonctions qui ont besoin d'un lien statique.

   Correction: On vient de voir que l'on se sert du lien statique pour deux choses : accès à une variable non locale, et appel d'une fonction ayant elle même besoin du lien statique. Soit F l'ensemble des fonctions nécessitant un lien statique :
   (a) si une fonction f fait un accès à une variable non locale, alors $f \in F$,
   (b) si une fonction f appelle une fonction $g \in F$, alors $f \in F$.

3. Donner un algorithme qui détermine l'ensemble de ces fonctions.

   Correction: Il suffit d'implémenter la caractérisation précédente. 1. Initialiser un ensemble F et un graphe de fonctions G à vide. 2. Parcourir l'AST en stockant dans F toutes les fonctions f demandant des variables non locales, et dans le graphe G un lien depuis toutes les fonctions qui sont appelées par f, et f elle-même. 3. Après avoir parcouru l'AST, faire une clôture transitive : pour toute fonction f dans F et toute fonction g telle que $(f, g) \in G$, mettre g dans F. 4. Répéter jusqu'à ce que F n'ait pas changé après une itération complète.

4. Dans le cadre du compilateur `tc` version 2004, où et comment l'implémenteriez-vous ?

   Correction: Dans l'`EscapeVisitor` : son constructeur initialise F et G, son parcours s'occupe de reconnaître les déclarations de fonctions, les accès à des variable non locales (pour F) et les appels de fonctions (pour G), enfin son destructeur s'occupe de la clôture.

3 "Displays"

Utiliser des liens statiques n'est pas la seule technologie qui permette de gérer les variables non locales, on peut également utiliser des "displays".

Un display est un (unique) tableau des "frame pointers" des fonctions (statiquement, dans le source) parentes. Par exemple, pour :

```
let function A () =
    let function B () =
        let function C () = ...
        in C () end
    in B () end
in A () end
```

lorsque `C` est appelée par `B`, elle-même appelée par `A`, le display contient :

```
3  FP de C
2  FP du dernier B appelé
1  FP du dernier A appelé
0  FP de main
```

parce que la "profondeur statique" de `C` est 3, celle de `B` 2 etc.

Le gestion du display se fait dans les prologues et épilogues des fonctions ; pour une fonction de profondeur statique i :

```
Sauvegarder le slot i du display
Installer le FP de cette fonction dans le slot i du display
# Corps de la fonction
Restaurer le slot i du display
```

1. Pourquoi prendre la peine de sauvegarder/restaurer le display ?

   Correction: Ne pas oublier le cas où l'on appelle des fonctions de même niveau que soi-même, comme tout simplement dans le cas de fonctions récursives. Quand on quitte l'appel le plus profond, on doit restaurer le frame des appels moins profonds.

2. Pourquoi prendre la peine d'installer le FP d'une fonction dans le display dès qu'on y entre, puisque l'accès à ses variables locales se fait sans l'aide du display ?

   Correction: Il est clair que le frame pointer doit être installé quand on appelle une fonction, donc si on ne le fait pas dans le prologue, on doit le faire avant tous les `call`. Dans ce cas, autant ne le faire qu'une seule fois, même si on prend le "risque" de le faire une fois au lieu de 0.

3. Soit le programme suivant :

   ```
   let var global := 69
       function global () : int = global
   in
     global ()
   end
   ```

   Montrer le code Tree (HIR) du **corps** de la fonction `global` quand on utilise les liens statiques.

   Correction: Grosso modo, ça va ressembler à :
   `move rv, mem (mem (temp fp) - 4)`
   car à `temp fp` (et non pas *dans* `temp fp`) on a stocké le static link qui pointe vers le frame de Main : `mem (temp fp)`. Dans le frame de `Main` `Global` est quelque part, mettons en `-4` par rapport au frame pointer de Main. Au total : `mem (mem (temp fp) - 4)`, ou peut-être même `mem (mem (temp fp))`. En tout cas, deux `mem`. Il ne reste qu'à faire le "`return foo`" : `move rv, foo`.

4. Même question avec un display.

   Correction: Grosso modo, ça va ressembler à ce qui suit, en supposant que le display soit stocké à l'adresse `temp display` :
   `move rv, mem (mem (temp display - 4))`
   car `global ()` est à une profondeur statique de 1, donc l'adresse de son frame est stocké dans la deuxième case du display (`mem (temp display - 4)`). Il ne reste plus qu'à y lire la variable `global` qui n'a plus de raison d'habiter en -4 du frame, puisque la case 0 n'est plus occupée par le lien statique.
   On acceptera :
   – le display qui pousse vers le bas : `move rv, mem (mem (temp display + 4))`,
   – avoir mis un offset pour `global` : `move rv, mem (mem (temp display - 4) - 4)`,
   – les deux : `move rv, mem (mem (temp display + 4) - 4)`,
   On n'accepte pas :
   – manque d'offset par rapport à `temp display`,
   – autre chose que deux `mem`.

5. Soit une variable `x` déclarée à une profondeur d et utilisée à une profondeur u. Combien faut-il d'instructions pour accéder à cette variable en utilisant les liens statiques ?

Correction: Pour chaque saut de frame pointer, il nous faut un `mem ()` de plus. À la profondeur 1, on avait :
`mem (mem (temp fp) - <offset>)`
à la 2, il faut :

```
mem (mem (mem (temp fp)) - <offset>)
```

toujours sous l'hypothèse que le lien statique est à l'offset 0 de chaque frame. On a donc, en négligeant le `binop`[a], on a donc $d + 1$ instructions. Notez que ce sont des instructions coûteuses : des lectures en mémoire.

a. Dans les assembleurs récents, faire `mem (t)` ou `mem (t + constante)` a le même coût.

6. Idem, en utilisant un display.

Correction: Le code vu deux questions au dessus se généralise en :
`mem (mem (temp display - (4 * d))`
sachant que `4 * d` peut se calculer à la compilation (d est connu). On a donc, quelque soit la profondeur, 2 instructions.

7. Combien d'instructions coûte la maintenance des liens statiques dans l'appel d'une fonction (on comptera le travail de l'appelant avant l'appel, le travail du prologue et de l'épilogue de l'appelé).

Correction: Pour l'appel d'une fonction, il faut calculer le lien statique de l'appelé. Si l'appelant et l'appelé sont à une distance de d (différence entre leur profondeur statique), alors calculer le lien statique coûte $d + 1$ instructions : c'est exactement la même chose que dans le cas exposé ci-dessus puisque le lien statique est stocké comme une variable à l'adresse 0 du frame.
Le prologue de l'appelé doit stocker le lien statique qu'on lui donne : une instruction.
L'épilogue de l'appelé ne fait rien.
Au total, en gros $d + 2$ instructions.
Toute réponse linéaire raisonnable ($d + 1$, $d + 3$ etc.) est acceptée.

8. One more time, mais avec les displays.

Correction: Pour l'appel d'une fonction, on n'a rien à faire.
Le prologue de l'appelé doit sauver la case du display où il va s'inscrire : 1 instruction. Il doit y stocker l'adresse de son frame : 1 instruction.
L'épilogue de l'appelé restaure la case du display qu'il occupait : 1 instruction.
Au total, en gros 3 instructions.

9. Qu'en concluez-vous ?

Correction: L'obtention d'une variable locale avec les liens statiques a un coût linéaire en la distance de profondeur. Du coup, l'appel de fonction aussi.
Les displays ont un comportement constant partout, avec un constante compétitive face aux liens statiques même à petite distance.
Les displays sont clairement plus attrayants. En plus, c'est plus facile à implémenter !
NB : aucune de ces études ne tient compte des optimisations possibles.