Correction du partiel CMP1 & TYLA

EPITA – Promo 2011
Tous documents (notes de cours, polycopiés, livres) autorisés
Calculatrices et ordinateurs interdits.

Janvier 2008 (1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain. Le *best of* est tiré des copies des étudiants, fautes de français y compris, le cas échéant.

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. Une lecture préalable du sujet est recommandée.

Merci de répondre sur votre copie, même pour les questions à choix multiples.

1 Passage d'arguments

1. Quel(s) type(s) de passage(s) d'argument(s) est (sont) utilisé(s) en C ?
   (a) passage par valeur/copie (*call by value*)
   (b) passage par référence (*call by reference*)
   (c) passage par résultat (*call by result*)
   (d) passage par nom (*call by name*)

   Correction: Seule la réponse 1a est valide : les autres types de passage d'arguments n'existent pas en C.

2. Même question avec C++ (attention, il y a un piège !).

   Correction: En sus du passage par valeur, C++ supporte le passage par référence grâce à &.

3. Même question avec Tiger.

   Correction: Tiger a la même sémantique de passage par copie que C et C++. Les valeurs complexes (tableaux, objets, etc.) sont manipulées par référence.

4. Il n'y a pas de type référence en Tiger ; comment peut-on les simuler avec des constructions du langage existantes ?

Correction: En utilisant des tableaux, des enregistrements ou des objets, puisque ceux-ci sont implicitement manipulés par pointeurs/référence en Tiger. Le programme suivant affiche `51` sur la sortie standard.

```
let
  type arr = array of int
  var a := arr[1] of 42
  var b := arr[1] of 0
in
  b := a;
  b[0] := 51;
  print_int(a[0]);
  print("\n")
end
```

Pour information, c'est d'ailleurs grâce à des enregistrements à champ unique que sont conçues les référence en Objective Caml :

```
# let a = ref 42;;
val a : int ref = { contents = 42 }
```

Attention, plusieurs copies mentionnent la possibilité d'utiliser des alias de type avec la construction `type` $t = u$ pour faire profiter t d'une sémantique de passage par référence que u n'aurait pas. Il n'en est rien : t est un synonyme de u, et de ce fait, il copie les comportements de u ; si u est un type passé par valeur (copie), alors t l'est aussi.

Best-of:
- On peut utiliser une variable globale.
- On peut utiliser le fait de déclarer une variable "globale" dans la fonction.
- On peut simuler avec des pointeurs.
- Avec les types en Tiger.
- On fait des copies.
- En compilant Tiger avec `g++` juste pour ces parties là.
- Avec le passage par adresse.
- Avec le mot clef `ref`.
- Utilisation du mot clef `primitive`.
- [...] à l'aide des variables imbriquées.
- On utilise le mot clef `new`.
- En faisant une union.
- Pourquoi vouloir simuler une chose qui n'existe alors que cela serais beaucoup plus simple de changer de langage si on souhaite vraiment cette caractéristique.

2 Const

À la différence de C++, Tiger est dépourvu de mot clef `const`. On se propose dans cet exercice de le rajouter dans le langage, en passant notamment en revue les différentes parties de notre compilateur afin de déterminer quels sont les aménagements nécessaires.

Rappel : le langage Tiger utilisé dans cet exercice est celui qui est décrit dans le Manuel de Référence du Compilateur Tiger, qui est utilisé comme référence tout au long du projet du même nom.

2.1 Préliminaires

1. À quoi sert `const` en C++ ? Il y a plusieurs réponses possibles, une seule est attendue).

   Correction:
   – Créer des constantes, au sens strict.
   – Créer des alias (via des pointeurs ou des références) de variables existantes, en restreignant leur accès à la simple lecture.
   – Appliqué à une méthode, garantir que le code de celle-ci ne modifiera pas l'objet cible.
   – Par extension des points précédents, permettre certaines optimisations (propagation de constantes, élimination de constantes, évaluation partielle, etc.).
   Attention, `const` ne sert pas à « figer » une zone mémoire comme certains l'ont écrit. Dans le code ci-dessous, `i` et `j` désignent la même zone mémoire, mais le fait que `j` soit qualifié de `const` ne donne aucune garantie sur la constance de `i` !

   ```
   int i = 42;
   const int& j = i;
   ```

   Best-of:
   – [...] Cela sert à gagner de l'espace mémoire qui devrait être utilisé pour une variable modifiable.
   – Le `const` sert à créer une méthode virtuelle.

2. Parmi les lignes suivantes, quelles sont celles qui sont valides en C++ ? Justifiez votre réponse.
   (a) `int i = 42; const int* const p = &i;`
   (b) `int i = 42; const int const * p = &i;`
   (c) `int i = 42; int const * p = &i;`
   (d) `int i = 42; int const * const p = &i;`

   Correction: Les symboles '&' ont sauté dans la version du sujet distribuée au partiel, désolé. J'ai ajusté la notation en conséquence.
   Seule la ligne 2b est incorrecte ; les autres sont juste. Rappelons la règle du placement du mot clef `const` : celui-ci s'applique à ce qui précède, sauf lorsqu'il est placé en tête ; dans ce cas, il qualifie ce qui suit. Dans le cas de la ligne 2b, les deux `const` s'appliqueraient donc tous les deux à `int`, ce que le compilateur refuse.

3. Même question.
   (a) `int i = 42; const int& const p = i;`
   (b) `int i = 42; const int const & p = i;`
   (c) `int i = 42; int const & p = i;`
   (d) `int i = 42; int const & const p = i;`

   Correction: Ici, une règle supplémentaire s'applique : une référence est implicitement un pointeur constant déguisé. En conséquence, `const` ne peut s'appliquer à &. Seule la réponse 3c est donc valide ici.

4. Soit la classe suivante :

```
struct C
{
  void m1 () {}       // (0).
  void m1 () const {} // (1).
  void m2 () {}       // (2).
  void m3 () const {} // (3).
};
```

Pour chacun des cas suivants, indiquer quelle méthode est appelée (0, 1, 2 ou 3) ou s'il y a une erreur de compilation, le cas échéant.

(a) C o; o.m1();

Correction: Réponse : 0 (o est non constant).

(b) **const** C o = C(); o.m1();

Correction: Réponse : 1 (o est constant).

(c) **const** C o = C(); o.m2();

Correction: Erreur de compilation : `m2` est une méthode non constante, et ne peut être appelée avec une cible constante (o).

(d) C o; o.m3();

Correction: Réponse : 3 (on peut passer un objet mutable à une méthode constante).

(e) **void** (C::*m)() **const** = &C::m1; (C().*m)();

Correction: Réponse : 1. Ici, `m` est un pointeur sur une méthode qui a la signature **void** (C::*)() **const**, initialisé avec l'une des méthode `m1` de `C`. Le `const` à la fin de la signature de `m` sélectionne la version constante de `m1`, d'où la réponse.

Best-of: Erreur de compilation et franchement si ça marche c'est honteux tellement ce code est hideux.

(f) **void** (C::*m)() = &C::m3; (C().*m)();

Correction: Erreur de compilation : C::m3 ne respecte pas le type du pointeur sur méthode `m`, qui ne se termine pas par un `const`, alors que c'est le cas de `m3`.

2.2 Partie frontale (*front end*)

Cette partie s'intéresse aux modifications à apporter au front end Tiger pour prendre en compte `const`. Nous ne considérerons ce nouveau mot-clef que comme qualificatif de types (le cas des méthodes « constantes » n'est pas à traiter).

1. **Spécification lexicales.** Que faut-il ajouter aux spécifications lexicales du langage pour supporter `const` ?

   Correction: Il suffit d'ajouter un nouveau mot-clef `const`. Le terme *token* ne convient pas ici : il s'agit de spécification, pas d'implémentation.

2. **Scanner.** Comment étendre le scanner, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`scantiger.ll`' pour supporter ces nouvelles spécifications lexicales ?

   Correction: Quelque chose de ce genre dans la section des règles lexicales (entre les deux occurrences de `%%`) :

   ```
   "const"          return token::CONST;
   ```

   Il faudra également penser à définir un nouveau token `CONST` dans '`parsetiger.yy`' (cf. réponse à la question 5).

3. **Spécification syntaxiques.** Que faut-il ajouter aux spécifications syntaxiques du langage pour supporter `const` ?

Correction: L'extension la plus simple et qui conserve tout le pouvoir d'expression de `const` (au détriment de constructions plus lourdes par endroits) consiste à augmenter la règle de production `type-id` pour qu'elle accepte les identifiants précédés de `const`.

type−id → id
type−id → *const* id

Certes, cela ne permet pas des écritures telles que :

```
type t = const array of int
```

mais il suffit de remplacer par :

```
type u = array of int
type t = const u
```

pour contourner cette limitation. Il bien sûr possible de supporter la première écriture, en modifiant un peu plus la grammaire, et au prix de quelques modifications pour lever les ambiguïtés introduites. Dans le reste de la correction, on considère que c'est l'extension la plus simple (la première décrite ci-avant) qui est adoptée.

Best-of: Obi-Wan Kenobi (*Ça faisait longtemps...*).

4. **Syntaxe abstraite.** Faut-il effectuer des modifications ou des ajouts dans la syntaxe abstraite du langage pour supporter `const` ? Si oui, lesquelles ?

Correction: Oui, il faut pouvoir matérialiser `const` dans la syntaxe abstraite. Plusieurs solutions s'offrent à nous.
- En ajoutant un attribut booléen (par exemple `const_`), à ast::NameTy, mis à `false` par défaut, et à `true` lorsque le type est précédé de `const`. Il faut équiper la classe : ajout d'accesseur(s), modification de constructeurs, etc.
- En ajoutant une classe ast::ConstNameTy, symétrique de ast::NameTy. Pour des raisons de factorisation, on peut au choix
  - faire que ast::ConstNameTy, dérive de ast::NameTy ;
  - faire dériver ces deux classes d'une même classe abstraite pour les mettre au même niveau dans la hiérarchie.

Pour des raisons de simplicité, nous choisissons la première solution pour le reste de la correction, mais toutes les réponses valides ont été acceptées.

5. **Parser.** Comment étendre le parser, c'est-à-dire, que faut-il ajouter/modifier dans le fichier '`parsetiger.yy`' pour supporter ces nouvelles spécifications syntaxiques ?

Correction: Les ajouts nécessaires sont :
- la définition du token `CONST`, déjà mentionnée plus haut ;
- la règle de production de la question précédente.

```
// ...
%token CONST       "const"
// ...
%%
// ...
typeid:
  ID      { $$ = new ast::NameTy (@$, take ($1), false); }
  // New rule.
| CONST ID { $$ = new ast::NameTy (@$, take ($2), true); }
;
// ...
```

où le constructeur de `ast::NameTy` est étendu pour prendre un booléen représentant la constance du type.

6. **Liaison des noms.** Que faut-il changer dans le `Binder` vis-à-vis de `const` ?

   Correction: Rien, à condition que les visiteurs vides (`DefaultVisitor`, `DefaultConstVisitor`) aient bien été écrits. En effet, nos modifications n'entraînent pas de changement vis-à-vis de l'existant.
   Certains proposent d'utiliser `const` lors de la résolution des appels de méthodes (qui peuvent être étiquetées `const` elles aussi). C'est une bonne idée, mais il faut se rappeler que Tiger est à l'origine un langage sans surcharge de fonctions/méthodes.

   Best-of: Des oursins, comme ça dès qu'on veut toucher à un `const`, on les jette à la figure de l'utilisateur, après je pense qu'il aura compris !

7. **Vérification des types.** Qu'allez-vous changer dans le module `type` du compilateur ?

   Correction:
   - Concernant les types du langage, le plus simple est d'équiper la classe type::Named d'un champ `const_`, à l'instar de ast::NameTy (cf. question 4).
   - Les affectations doivent être vérifiées : une valeur de type constant ne peut être une l-value. La méthode type::TypeChecker::**operator**() (ast::AssignExp& e) doit donc s'assurer que l'opérande gauche de l'opérateur := est non constant.
   - Il faut aussi propager la constance. La copie de valeurs (r-values) constantes lors d'une initialisation (définition de variables ou d'arguments formels) ou affectation d'une l-value du type non constant doivent être interdites, au moins en ce qui concerne les valeurs manipulées par référence (tableau, enregistrements, etc.). En effet, on veut interdire ce genre de manipulations :

```
let
  type rec = { x : int }
  type const_rec := const rec
  var a : const_rec := rec { x = 42 }
  var b : rec := a  /* Danger: 'b' is a mutable
                       alias of 'a' ! */
in
  /* a.x := 51 */  /* Forbidden, 'a' is const. */
  b.x := -42        /* Allowed: 'b' is mutable.
                       => 'a.x = -42' !! */
end
```

   Best-of: La collecte d'oursins, pour approvisionner le Binder.

8. **Représentation intermédiaire** Faut-il ajouter/modifier quelque chose dans le langage Tree ?

   Correction: Réponse courte : non. En effet, une fois passée l'analyse sémantique, `const` n'a plus de rôle à jouer.

   Réponse longue : on pourrait cependant décider d'*annoter* les sous-arbres correspondant aux nœuds étiquetés par un type `const` dans l'AST, afin d'autoriser certaines optimisations dans le *middle end*. Il aussi est possible d'effectuer ces optimisations en amont (avant la traduction), notamment pour des raisons de simplicité ; cependant, les appliquer sur la représentation intermédiaire présente un intérêt : ces optimisations sont indépendantes des langages sources et cibles du compilateur, et donc réapplicable dans d'autres contextes que la compilation Tiger → MIPS (ou Tiger → IA-32).

Best-of:
- La table des symboles [...] sera plus lourde.
- On devra ajouter le mot `const` dans la représentation intermédiaire en langage Tree. (*Il existe déjà !*)

9. **Traduction (vers IR).** Y'a-t-il des changements à apporter au visiteur chargé de la traduction vers la représentation intermédiaire ? Justifiez votre réponse.

Correction: Non, cf. réponse précédente.

Best-of:
- Au moment où de la traduction vers IR, cette vérification a déjà été faite et les variables const sont strictement identiques aux variables normales (j'allais dire "variables variables" mais ça irait direct dans le best off :( ).
- Peut-être que oui ou peut-être que non... Bonne question.
- Non. Ma réponse est basée sur une probabilité que la réponse soit non (1 chance sur deux).
- La réponse D.
- Le '`ChangeLog`', '`README`' et l'`help` ;P puisque la grammaire a changé.
- Je dirais que non vu la tournure de la phrase.

2.3 Partie terminale (*back end*)

Nous n'avons pas étudié le back end de `tc` à ce stade de l'année, mais nous avons déjà un évoqué ses composants, et vous avez déjà été sensibilisés à des notions connexes dans d'autres cours (assembleur, architecture des ordinateurs, etc.).

Grossièrement, le back end de `tc` effectue deux tâches :

L'allocation de registres Chaque fonction peut contenir un nombre arbitraire de variables ou *temporaires* ; or un processeur n'a qu'un nombre fini de *registres*. L'allocateur de registres a pour tâche de résoudre ce problème en faisant appel à la mémoire au besoin.

La génération de code en langage d'assemblage Il s'agit de la dernière étape de traduction, qui fait le pont entre la représentation intermédiaire (en Tree) et la sortie (en langage d'assemblage MIPS ou IA-32 dans notre cas).

1. **Allocation de registres.** Est-ce que l'adjonction de `const` à Tiger modifie l'allocateur de registres ?

Correction: Non : `const` est du ressort du front end, et son rôle s'arrête après l'analyse sémantique.

Best-of:
- Oui. Il faut spécifier a l'allocateur de registres que les variables déclarer constante ne doivent pas être chargées dans les registres du processeur (c'est le but).
- [...] le back end n'est pas concerné par "const". Il s'en fiche. Il l'ignore royalement en allant à la piscine le jeudi.
- `const` est un identifiant. [...] il fera appel à la mémoire au besoin.
- Il faut utiliser des registres spéciaux pour les variables const afin de les distinguer par la suite (de ceux à utiliser en lecture/écriture).
- Une fois de plus, la partie front-end s'est chargée de résoudre tout problème concernant la modification inopinée d'un const. Après certains dirons : "on est jamais trop sur"...
- [...] une variable constante (ça fait un peu pléonasme)
- Oui, il faudra stocker les constantes dans des registres constants.

2. **Génération de code.** De même, la génération de code à partir de la représentation intermédiaire est-elle affectée ?

   Correction: Non, pour les mêmes raisons qu'à la question précédente.

   Best-of: La génération de code à partir de la représentation intermédiaire est bien évidemment affecté, si par exemple on génère du code en C++, on emploiera le `const` de C++.