Correction du partiel CMP1 & TYLA

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

(1h30)

Correction: Le sujet et sa correction ont été écrits par Roland Levillain.

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.

1 Énumérations fortement typées

Dans cet exercice, on s'intéresse à l'ajout d'énumérations (mot clef `enum` en C++) fortement typées dans le langage Tiger. Nous allons notamment passer en revue les différentes parties de notre compilateur afin de déterminer 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.

1.1 Préliminaires

1. À quoi peuvent servir les énumérations ? Donnez un ou deux exemples.

   Correction: Créer des types « ensembles » exhaustifs avec relativement peu de valeurs, et distincts des autres types (que ce soient des types énumérés ou pas). Des exemples classiques sont des énumérations représentant les jours de la semaine, les noms des mois, des noms de *tokens*, les booléens, etc.

2. Quel est le défaut des énumérations en C++ ?

   Correction: Elle sont très faiblement typées, puisque les étiquettes d'une `enum` sont implicitement convertibles en entiers. Au point que deux étiquettes de deux énumérations différentes peuvent être confondues ! Pour information, le code suivant compile sans aucun warning, malgré un niveau de vérification élevé (option `-Wall` et `-Wextra` de `g++`).

Correction:

```
enum color  { red, black };
enum planet { mars, jupiter };

int main ()
{
  color c = red;
  switch (c)
    {
    case red:
      return 1;
      break;
    case jupiter:
      return 2;
      break;
    }
}
```

3. Avant d'étendre le compilateur, réfléchissons à la forme que nous souhaiterions donner à cette extension. Donnez un exemple court mais complet de ce que vous souhaiteriez pouvoir écrire avec une telle extension pour les énumérations. Votre exemple devrait au moins comporter
   – une déclaration d'un type énumération ;
   – une déclaration d'une variable de ce type ;
   – une utilisation de cette variable.

Correction: Par exemple :

```
let
  type gesture = enum { rock, paper, scissors }

  function best_gesture (g1 : gesture, g2 : gesture) :
    gesture =
  if g1 = rock then
    if g2 = paper then paper else rock
  else if g1 = paper then
    if g2 = scissors then scissors else paper
  else /* g1 = scissors */
    if g2 = rock then rock else scissors

  var my_move := scissors
  var their_move := paper
in
  best_gesture (my_move, their_move)
end
```

1.2 Partie frontale (*front end*)

Cette partie s'intéresse aux modifications à apporter au front end Tiger pour supporter les énumérations.

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

Correction: En se basant sur ce qui est proposé dans la réponse à la question 3, on constante qu'il suffit d'ajouter un nouveau mot-clef `enum` ; les autres éléments lexicaux font déjà partie du langage.

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 `%%`) :

   ```
   "enum"          return token::ENUM;
   ```

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

3. **Spécification syntaxiques.** Que faut-il ajouter aux *spécifications* syntaxiques du langage pour supporter les types énumérations ?

   Correction: Il faut
   (a) en premier lieu étendre la règle de production `type-id` pour qu'elle accepte les déclarations d'énumérations, par exemple :

   ty → 'enum' '{' labels '}'

   (b) puis introduire une règle permettant de définir une liste d'étiquettes (non vide) :

   labels → ID { ',' ID }

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

   Correction: Bien entendu. Il faut fournir un nouveau nœud d'AST pour les type énumérés, par exemple `EnumTy`, ayant pour attribut une liste d'identifiants.

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: Nous devons :

– augmenter l'`union` de Bison pour qu'elle accepte une liste d'identifiants comme valeur sémantique (liste d'étiquettes d'une énumération) ;

```
namespace ast
{
  typedef std::list<misc::symbol*> labels_type;
}

%union
{
  ast::labels_type* labels;
}
```

– ajouter un token `ENUM` ;

```
%token ENUM "enum"
```

– annoncer le symbole non terminal `labels` et sa valeur sémantique associée (une liste d'identifiants) ;

```
%type <labels> labels
```

– implémenter les règles de production de la question 3.

```
ty:
  ENUM "{" labels "}"  { $$ = new ast::EnumTy (@$, $3); }
;

labels:
  ID                { $$ = new ast::labels_type;
                      $$->push_back ($1); }
| labels "," ID       { $$ = $1; $$->push_back($3); }
;
```

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

Correction: Potentiellement, rien, à condition que les visiteurs vides (`DefaultVisitor`, `DefaultConstVisitor`) aient bien été écrits. Les nouveaux noms introduits (noms de types énumérés, noms d'étiquettes) sont des identifiants, déjà pris en charge par le `Binder`.
S'agissant de la sémantique de redéfinition des étiquettes, on a cependant le choix parmi plusieurs politiques :
– interdire toute réutilisation d'étiquettes dans une définition d'`enum` au sein d'un programme entier ;
– interdire toute réutilisation d'étiquettes dans une définition d'`enum` au sein d'un même bloc (*chunk*) de définition de types ;
– autoriser la réutilisation d'étiquettes tant que celui ne comporte aucune ambiguïté. Par exemple :

```
type eye   = enum { blue, green, brown }
type light = enum { red, green, blue }

var a : eye := blue     /* OK. */
var b : light := green  /* OK. */
var c := blue           /* Forbidden, ambiguous. */
```

– autoriser la réutilisation d'étiquettes tant que celui ne comporte aucune ambiguïté, et fournir un moyen de désambiguïser le cas échéant. En reprenant l'exemple précédent :

```
var d := eye.blue       /* OK, not ambiguous. */
```

Dans ce cas, il faut bien entendu modifier (un peu) la grammaire pour supporter cette syntaxe avec un nom d'étiquette « pleinement qualifié ».
Tout comme pour la liaison des champs d'enregistrements et des membres de classes, on peut décider de laisser au `TypeChecker` le soin d'effectuer ces opérations sur les étiquettes (cf. question suivante).

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

Correction: Nous devons introduire une classe `type::Enum` pour matérialiser le type d'une nouvelle énumération dans le compilateur.
L'information la plus importante est le lien entre une étiquette et l'énumération qui la possède. Cette étape peut être effectuée par le `Binder` comme par le `TypeChecker`. La suite des opérations est immédiate : la visite d'une `ast::EnumTy` construit un `type::Enum` ; une `ast::SimpleVar` (variable nommée) faisant référence à une étiquette récupère le type de son énumération en vertu du lien précédemment mentionné. Le reste des règles de typage (vis-à-vis de l'initialisation et de l'affectation notamment) est déjà implémenté dans le `TypeChecker`.

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

Correction: Rien : une fois l'analyse sémantique effectuée, on peut sans crainte (modulo les bugs du compilateur !) traduire les valeurs d'énumérations en entiers (comme en C++), et les manipuler en tant que tel. Tree est suffisant à cet égard, et ne nécessite pas d'ajout.

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: Comme mentionné à la question précédente, l'approche la plus simple est d'étendre le `Translator` pour traduire une `ast::Simplevar` étiquetée par un type `type::Enum` en une valeur entière, correspondant à la position de l'étiquette dans l'énumération.

1.3 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 d'énumérations à Tiger modifie l'allocateur de registres ?

   Correction: Non : les types sont ressort du front end, et une fois passée l'analyse sémantique, les valeurs énumérées ne sont plus considérées que comme des entiers.

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.

1.4 La question bonus

Quelle extension concomitante de celles des énumérations pourrait-on souhaiter dans Tiger ?

Correction: Celle de `switch`, pardi ! En effet, `switch` est particulièrement bien adapté aux énumérations, car l'exhaustivité des valeurs de l'aiguillage peut aisément être vérifiée par le compilateur, lorsqu'elles appartiennent à une même `enum`.

2 Booléens

Dans cette partie, on étudie une application des types énumérés implémentés dans la section précédente, concernant les types booléens.

1. Les booléens peuvent être vus comme des types énumérés. Quel peut-être l'intérêt de les fournir dans la bibliothèque standard du langage sous forme d'énumération (plutôt que de les confondre avec les entiers, comme c'est le cas actuellement) ? Attention, il ne s'agit pas d'une extension supplémentaire (en sus de celle des énumérations).

   Correction: Encore une fois, cela permet une vérification des types plus strictes : au sens mathématique, un valeur booléenne n'est *pas* un entier. Les langages de programmation sont parfois laxistes sur ce point (essentiellement pour des raisons de facilité d'écriture pour l'utilisateur et/ou vis-à-vis de implémentation de compilateurs).

2. Qu'ajouteriez vous (types, fonctions, etc.) au prélude de `tc` pour étendre la bibliothèque standard avec un type énuméré représentant les booléens ? Vous pouvez soit fournir du code annoté, soit décrire votre solution sous forme de description.

Correction:

```
/* The type of Booleans. */
type bool = enum { true, false }

/* Logical and. */
function and (b1 : bool, b2 : bool) : bool =
  if b1 = true & b2 = true then true else false

/* Logical or. */
function or (b1 : bool, b2 : bool) : bool =
  if b1 = false & b2 = false then false else true

/* Conversion from bool to int. */
function int_of_bool (b : bool) : int =
  if b = true then 1 else 0

/* Conversion from int to bool. */
function int_of_bool (i : int) : bool =
  if i = 0 then false else true

/* Conversion from bool to string. */
function string_of_bool (b : bool) : string =
  if b = true then "true" else "false"

/* ... */
```

3. Quel est le défaut de cette approche ?

Correction: Essentiellement le manque d'intégration avec le reste du langage. C'est souvent le prix à payer pour une extension peu ou pas intrusive.
Un prédicat comme

```
is_prime(i : int) : bool = { ... }
```

ne s'intègre pas directement dans un test (`if`), puisque ceux-ci attendent un entier ; ironie du sort, une conversion explicite devient nécessaire :

```
if int_of_bool(is_prime(42)) then "prime" else "composite"
```