Partiel de Compilation : Surcharge

EPITA – Promo 2005 – Documents autorisés

Avril 2003

> Une copie concise, bien orthographiée, dont les résultats sont clairement exposés, sera toujours mieux notée qu'une copie qui aura demandé une quelconque forme d'effort de la part du correcteur.

Un soucis constant de l'inventeur de langage est la conception de fonctionnalités :

orthogonales qui peuvent s'utiliser librement sans contraintes artificielles dues aux interactions entre fonctionnalités

ouvertes à l'utilisateur la bibliothèque du langage n'a pas plus de droits que l'utilisateur

implémentables garder en tête qu'il faudra un jour le mettre en œuvre dans un compilateur. Dans le cas d'extensions d'un langage, il est souhaitable de plus qu'elles soient

conservatives que les programmes valides du langage souche le restent dans le langage étendu.

Nous allons nous intéresser à certaines formes de *polymorphisme*, interdites au programmeur Tiger, et nous efforcer de l'ouvrir, en tâchant de respecter les critères précédents.

1 Polymorphismes

1. Qu'est-ce que le polymorphisme ? On s'attachera à donner une définition générale, et non pas liée à une forme particulière de polymorphisme, ou bien à un langage spécifique.
2. Soient `a` et `b` deux variables Tiger. Donner une expression Tiger dont le code effectif (c'est-à-dire celui qui sera exécuté) dépend du type de `a` et `b`.
3. Comment le compilateur détermine-t-il la version du code à générer ?
4. Dans l'expression C `sin (51)`, quel autre mécanisme de polymorphisme intervient ?
5. Voyez-vous en Tiger une valeur qui puisse avoir plusieurs types ?
6. Les langages orientés objets introduisent une forme de polymorphisme. Comment l'appelleriez-vous ?
7. À quel mot-clé est-il associé en Simula et en C++ ?
8. Sur quelle forme de polymorphisme s'appuie largement STL ?

Quatre formes de polymorphisme viennent d'être exhibées. Nous nous intéresserons aux deux premières, celles que l'on rencontre déjà dans Tiger, mais qui sont interdites à l'utilisateur.

2 Tiger : Remise en Jambes

1. Rappeler quelles sont les grandes phases du compilateur Tiger. On demande un court schéma présentant des boites dont les entrées et sorties sont liées.
2. Expliquer le déroulement de l'analyse de type des déclarations de fonctions en Tiger.
3. Expliquer le déroulement de l'analyse de type des appels de fonctions en Tiger. Insister sur les différents cas d'erreur possibles.

3 OverTiger : Surcharge en Entrée

On souhaite étendre le langage Tiger pour qu'il supporte une forme de surcharge des fonctions comparable à celle du C++. Appelons OverTiger ce langage, dont les primitives sont les mêmes que celles de Tiger (`print`, `print_int` etc.).

1. Discuter la possibilité d'implémenter la surcharge au sein de l'analyse syntaxique.
2. Discuter la possibilité de gérer la surcharge dans le back-end.
3. En montrant où sont les différences avec Tiger, en déduire que l'implémentation la plus naturelle de la surcharge se fait pendant l'analyse de type. On expliquera en particulier pourquoi il est inadapté de vouloir résoudre la surcharge après avoir effectué la phase de contrôle de type.
4. Quel message d'erreur pourrait-on vouloir émettre sur le programme `print (51)` ?
5. En déduire l'ensemble des changements à apporter dans le `TypeVisitor` de Tiger pour l'adapter à OverTiger. On n'oubliera pas de parler des objets outils qui servent dans le contrôle de type.
6. Dans le cas d'un compilateur qui supporte les deux langages Tiger (`tc -x tiger`) *et* OverTiger (`tc -x overtiger`), expliquer comment factoriser le contrôle de type.

4 AboveTiger : Surcharge en Sortie

On s'intéresse à une extension de OverTiger, AboveTiger, qui prenne aussi en compte la surcharge sur les valeurs de retour des fonctions. Les primitives sont les mêmes que celles de AboveTiger, donc de Tiger (`print`, `print_int` etc.).

1. Montrer que sans aucune restriction, AboveTiger serait un langage mal défini, puisqu'un programme pourrait donner lieu à deux comportements différents.
2. Formuler une restriction générale qui interdirait de tels programmes.
3. Montrer que cette extension, à cause de la restriction que l'on vient d'introduire, n'est pas conservative.
4. Montrer comment mettre en œuvre cette restriction en expliquant le contrôle de type des appels de fonctions.

5 BeyondTiger

Si jamais vous estimez ne pas avoir eu suffisamment de place pour démontrer vos facultés de concepteur de langage, vous êtes invité(e) à décrire ici comment vous envisagez d'aller encore plus loin dans la surcharge qu'AboveTiger.

6 Exemples de Programmes

Cette section a double vocation : présenter quelques exemples de programmes Tiger qui pourraient réveiller les souvenirs de ceux qui auraient peu travaillé sur `tc`, mais aussi présenter des exemples des extensions étudiées pendant cet examen. Vous êtes chaudement encouragés à faire référence à ces exemples dans votre copie.

6.1 Tiger : Factorielle

```
let function fact (n : int) : int =
    if n then n * fact (n - 1) else 1
in
  fact (51)
end
```

6.2 Tiger : R'n B (Beurk)

```
let type red   = {h : int,    t : black}
    type black = {h : string, t : red}
    function print_red (red~: red) =
      if red != nil then
        (print_int (red.h); print_black (red.t))
    function print_black (black : black) =
      if black != nil then
        (print (black.h); print_red (black.t))
in
  print_red (red {h = 123, t = black {h = "456", t = nil}})
end
```

6.3 Tiger : Masquage

Masquage des primitives `print` et `chr`.

```
let function print (i : int) = print_int (i)
    function chr   (s : string) : string = s
in
  print (chr ("a"))
end
```

6.4 OverTiger : R'n B (OverBeurk)

```
let function print (i : int) = print_int (i)
    type red   = {h : int,    t : black}
    type black = {h : string, t : red}
    function print (red : red) =
      if red != nil then
        (print (red.h); print (red.t))
    function print (black : black) =
      if black != nil then
        (print (black.h); print (black.t))
in
  print (red {h = 123, t = black {h = "456", t = nil}})
end
```

6.5 OverTiger : Id

```
let function id (s : string) : string = s
    function id (i : int)    : int    = i
in
  (id ("1") = "1") = (id (1) = 1)
end
```

6.6 AboveTiger : Read

En admettant l'existence des primitives `read_string` qui lise l'entrée standard et retourne une chaîne de caractère, et `read_int` qui fasse de même pour un nombre.

```
let function read () : int    = read_int ()
    function read () : string = read_string ()
in
  print_int (read ());
  print     (read ())
end
```

6.7 AboveTiger : AB

```
let type a = {}
    type b = {}
    function ab () : a = a {}
    function ab () : b = b {}
    function  b () : b = b {}
    function foo (x : a, y : a) = print ("aa\n")
    function foo (x : a, y : b) = print ("ab\n")
    function foo (x : b, y : a) = print ("ba\n")
in
  foo (ab (), b ())
end
```