Typages et programmation - Option

Simuler de l'objet en C (modularité)

Encapsulation des données, constructeur…

Programmation modulaire en C : *diviser le programme en modules qui intérragissent*, elle se fait ici à travers 2 fichiers : - Le `.h` : le header, l'information publique du module (en objet on appelle ça une interface) - Le `.c` : l'implémentation de l'interface

Avantage : Le monde extérieur ne connaît que ce qui est dans le `.h`. On peut donc modifier son implémentation à volonté.

Par exemple : Les modules, `string`, `stack`…

Écrire un module de vecteur extensible

pseudo code

```
/* vecteur.h */
#ifndef v_H
#define v_H

typedef struct vecteur *vecteur;
void liberer(vecteur v);
vecteur creerVecteur();
void *get(vecteur v, int i);
void set(vecteur, int i, void *e);
int length (vecteur v);
#endif
```

```
/* vecteur .c */

#include "vecteur.h"

struct vecteur {
    void **tab;
```

```
    void (*liberer)(void *);
    int taille;
};

vecteur creerVecteur(/*?*/ void (*lib)(void *)) {
    vecteur v = malloc(sizeof(vecteur));
    v->tab = malloc(5);
    v->taille = 5;
    v->liberer = lib;
    return v;
}

void *get(vecteur v, int i) {
    if (i > v->taille-1) {
        printf("erreur");
    }
    return v->tab[i];
}

// rappel on ne gère pas toutes les erreurs dans ce code
int length(vecteur v) {
    return v->taille;
}

void set(vecteur v, int i, void *e) {
    if (i < v->taille) {
     v->tab[i] = e;
    }
    else {
            v->tab = realloc(2*v->taille);
            v->tab[i] = e;
    }
}

void liberer(vecteur v) {
    void (int i = 0; i < v->taille; i++) {
        v->liberer(get(v,i));
    }
    free(v->tab);
    free(v);
}
```

Chapitre 1 : Introduction au typage

On voudrait vérifier certaines propriétés :

- terminaison : est-ce que le programme tourner indéfiniment ? C'est une question non décidable en général.
- correction / vérification : est-ce que le programme vérifie certaines propriétés :
  - accès à des pointeurs NULL.
  - accès à une case non existante d'un tableau.

Globalement non décidable mais une sous-partie non négligeable (% à nos besoins) est décidable.

Pour ces vérifications, on a plusieurs méthodes :

- analyse statique (ex : à la compilation)
- analyse dynamique à l'exécution (des tests, monitoring à l'exécution avec *gdb* par exemple ou outil de vérification formelle, *coq*)

Les langages de programmation ne se distinguent pas par leur expressivité. → Ce que l'on peut écrire avec le langage *L1* on peut le faire avec le langage *L2*

On les différencie par leur capacité à faciliter le travail du programmeur : Le typage (types et systèmes de types) est un des moyens de les différencier. Il n'y a pas beaucoup de solutions : il faut rester dans un cadre décidable. Ensuite statique ? dynamique ? fort ? faible ? etc.

2 références sur le typage :

- *Pierce B.C Types and Programming languages MIT Press 2002* (plus facile)
- *Cardelli : Types Systems CRC Press 1997*

Un langage non typé basé sur les machines de Turing

Un programme dans une machine de Turing est un ensemble de quadruplet (*qi, sj, Ah, qk*) *qk* état après avoir exécuté l'instruction

- *qi* est la valeur d'un état
- *sj* est un symbole parmi un ensemble S (on va dire {0,1}) qui sont lus sur une bande
- *Ah* est un symbole {→, ←} (déplacer la tête à droite ou à gauche)
- On dispose d'une instruction initiale (*q0, s0, A0, q'0*)

Le programme s'arrête lorsque aucune règle ne peut-être appliquée.

On sait d'après Turing (36) que toutes les fonctions décidables admettent un programme dans ce langage.

Testez combien c'est verbeux avec la multiplication de suites binaires dans un tel langage.

Deuxième langage non typé : -calcul pur (Church)

Syntaxe

on a : - V : un ensemble de variables - Σ : V U { , (,), .}

Un -terme c'est : - x appartenant à V. - x appartenant à V, M un -terme, alors x.M est un -terme - M1 et M2 sont des -termes, alors (M1 M2) est un -terme

exemple : x.x, fonction identité. (x x). x. x.x, ceci va questionner sur l'ordre d'évaluation

Comment on peut coder les entiers

(Par induction)

- 0 c'est f. x.x
- 1 c'est f. x.(f).x
- 2 c'est f. x.(f)(f).x

Dans de tels langages non typés, certaines questions se posent rapidement : * Je veux écrire une fonction qui ne manipule qu'un certain nombre d'objets (exemple : -terme représentant les entiers). Comment faire ? * Comment être certain que vous renvoyez bien un entier ? * Comment distinguer les fonctions ? * Comment bien représenter les objets ? but : faciliter le calcul

→ Le typages c'est entre autres essayer de répondre à ce type de questions.

Le typage permet de vérifier une cohérence entre des ensembles de valeurs et le comportement souhaité.

Pour des soucis d'efficacité des choix ont été faits sur la représentation des valeurs (ceci facilite par exemple la génération de code)

Définition

Un type est constitué : - D'un ensemble de valeurs (E) - Un ensemble d'opérations sur les valeurs de E (les opérations définissent les propriétés)

exemple : (int, (+,-,/,%,<=,>=))

Un langage est dit *typé* si à certains de ces éléments on associe un type.

Question de base

Comment identifier (se réduit) les objets typables ? (environnement + élément typable) : peut-on déduire son type ?

Le système de type d'un langage c'est * un ensemble de types * un ensemble de règles permettant de répondre (ou partiellement) à la question de base

Dans un type on distingue : - la description mathématique du type (TDA, comportements des fonctions sur les valeurs) : *version abstraite* - l'implémentation (comment on représente les valeurs, l'implémentation concrète des opérations) *version concrète*

La représentation (ou implémentation) concrète peut être réalisée * au niveau matériel (circuit faisant les opérations pour les entiers) * au niveau logiciel (en manipulant d'autres types et les opérations seront souvent des fonctions)

Exemple :

- *Entiers* : (Z, + : Z × Z → Z)
  Comment "+" opère sur les 2 valeurs est spécifiée par la représentation mathématique du type. ("×" ici produit cartésien).
- *Pile* : (Pile, Objet, empiler : Pile × Objet → Pile, dépiler : Pile → Pile, sommet : Pile → Objet)

Lorsque l'on a une description du type, on est en mesure de cacher l'implémentation. Lorsque l'on donne une spécification on la voudrait non ambiguë (pas plusieurs possibilités)

La définition mathématique sera basée sur *l'algèbre* avec des opérations ensemblistes. En général on a au moins deux types d'ensembles dans la définition du type : * l'ensemble représentant les données * l'ensemble représentant l'objet à définir

Exemple : Pile

- Ensemble des objets que l'on stocke.
- Ensemble des piles.

Chaque opération il faudra dire si elle s'applique à l'ensemble des données ? à l'ensemble de l'objet à définir ? aux 2 ?

Exemple : Listes

(et variantes pile, file, …)

- Arbres

Vu que pour 1 type on a peut-être besoin d'autres types, il y a une hiérarchie dans les types : * Les types de base (Ex en C : `int, float, char, double`) * les types construits (ou composites) à partir de types existants (Ex en C : les tableaux, `struct`, `enum`). Ils sont obtenus à partir d'opérateurs (donnés avec le langage). * Les types proposés par le langage et différents des… (Ex: en C++ `bool`)

On fait la différence entre les types proposés et les types de base qui sont des types atomiques, des autres types proposées par le langage qui peuvent être obtenus par construction à base d'autres types.

Exemple de constructeurs pour des types composés :

- enregistrements (`struct, union` en C)
- modules (lanages ML)
- classes (Java, C++,…)
- fichiers (comme encapsulateurs)
- fonctions (langages fonctionnels)

Propriétés des systèmes de types

3 façons de caractériser les systèmes de types :

- statique ou dynamique
- les types sont explicites ou implicites
- le système est faible ou fort (avec des degrés)

Nos fonctions pour avoir du contrôle sur les entrées ou les valeurs manipulées, on donne des types aux entrées et aux valeurs manipulées. Le système de types pour contrôler la validité des types a deux stratégies :

- **statique** : le contrôle est fait pendant la compilation (langages dits *statiques*)
- **dynamique** : les contrôles sont faits pendant l'exécution, ceci suppose que les valeurs sont annotés avec le type pour vérifier la cohérence avant utilisation

On parlera de langage dynamique lorsque ce dernier a une stratégie dynamique. Certains langages peuvent mélanger les 2 stratégies.

Exemple de langage dynamique : Scheme

(version guile)

Fonctionnel et dérivé du Lisp. C'est un langage interprété.

```
scheme@(guile-user)> (define (f x) (+ x "a"))
scheme@(guile-user)> (f 3)
<unnamed port>:1:14: In procedure f:
<unnamed port>:1:14: In procedure +: Wrong type: "a"

scheme@(guile-user) [2]> (define (f y) (* g y))
;;; <stdin>:2:14: warning: possibly unbound variable `g'
scheme@(guile-user) [2]> (f 3)
<unnamed port>:2:0: In procedure f:
<unnamed port>:2:0: In procedure module-lookup: Unbound variable: g
```

Avantages de la stratégie statique

- détection : des erreurs potentielles liées aux types avant l'exécution (évaluation).
- documentation des erreurs liée aux types.
- parfois le compilateur permet d'optimiser le code.

Explicite vs Implicite

Un système de types est dit explicite si les types doivent être déclarés. Sinon il est implicite

- Implicite : C, Java, …
- Explicite : Python, OCaml, Javascript, Scheme

Un langage est implicite et statique doit avoir un moyen de vérifier les types à la compilation. Ce mécanisme c'est l'inférence de types.

Souvent explicite et statique vont ensemble (Ada, C, Java), implicite et dynamique aussi (SmallTalk python), **Mais pas toujours** famille ML implicite et statique

Système fort vs faible

Un système statique est dit **fort** si aucune erreur de type ne se produit pendant l'exécution. Un langage qui contient un système de type Fort est dit fortement typé. Tout langage non fortement possède un système de type dit **faible**.

Vue ensembliste

Pok = l'ensemble des programmes syntaxiquement corrects

Psafe = ceux à l'exécution sans erreurs de type

Fort <=> Pok = Psafe Faible <=> Pok Psafe != ø

Si votre langage a la même puissance que la machine de Turing => il ne peut pas être fort

Ex : On ne peut pas détecter dans x/y que y != 0.

Les faibles seront classifiés par rapport aux types *d'erreur* :

1. Les erreurs non détectées à la compilation comme à l'exécution. (C, C++)
2. Avortement brutal sans information.
3. Déclenchement d'exceptions pour permettre aux programmeur de récupérer l'erreur. (Ocaml, Java)
4. Le compilateur refuse l'expression sans information.
5. Refus de l'erreur et indication.

Un système de type est d'ordre j si les programmes engendrent des erreurs au plus i.

L. Cardelli : Considère un type fort ceux d'ordre au moins 2.

EXO

C

Un ou plusieurs exemples d'ordre 1.

Accès à une case non existante d'un tableau

```c
#include <stdio.h>

int main() {
    int tab[3] = {0, 1, 2};
    printf("%d\n", tab[4]);
    return 0;
}
```

Les *casts* en C sont physiques et non logiques.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = malloc(sizeof(int));
    *p = 3;
    void *r;
    float *f;
    r = (void *)p; // transition non nécessaire
    f = (float *)r;
    printf("%d, %g\n", *p, *f);
    return 0;
}
```

Java

Un ou plusieurs exemples d'ordre 3

Hiérarchie des types. `Object` est le type *racine* en java. Le cast n'est pas physique mais logique en java.

```
class Test {
    public static void main(String[] args) {
        Integer p = new Integer(5);
        Object o = (Object)p;
        Double d = (Double)o;
    }
}
```

(...) ?

Polymorphisme

Lorsqu'un objet peut admettre plusieurs types on dit qu'il est polymorphe.

Le mécanisme dérrière cette propriété est appelé *polymorphisme*.

Exemple : Un fonction polymorphe est une fonction dont les *paramètres* et le *résultat* peuvent être des instances de plusieurs types différents.

Remarque : Un système de types dynamique induit un certain polymorphisme naturellement.

On va distinguer plusieurs sortes de polymorphismes

Polymorphisme ad-hoc

Un objet ne peut accepter qu'un ensemble fini de types distincts.

Pas une seule façon d'implémenter ce polymorphisme. Le choix du type d'un objet peut être soit automatisé (intégré dans le système de types) soit laissé à la charge du programmeur.

```
struct point
{
    float x,
    float y
};

union z {
    int i,
    char c,
    struct point p,
```

```
};

/* ... */
```

Polymorphisme de surcharge

Un même *identificateur* peut représenter un nombre fini d'éléments distincts du programme c'est une facilité syntaxique. Le choix du bon objet est fait par le système de types *à la compilation* (résolution statique).

Question :

```
int f() {
    /* code */
}

char f() {
    /* code */
}
```

Impossible en C. Car le compilateur ne saura pas quelle fonction choisir à la compilation.

/! En général l'implicite et la surcharge ne vont pas ensemble.

Exemple en Ocaml :

`+` pour les `int` `+.` pour les `float`

Polymorphisme de coercition

Le type *t1* est **transformable** au type *t2*. Si toute instance de *t1* peut être transformée en une instance de *t2*.

Une transformation est faite par le système de types implicitement. Le polymorphisme de coercition = ensemble fixé (et fini) de transformations entre types (dans le but de se conformer à un contexte).

La résolution par le système de types statiquement.

Exemple : en C : `4+5.` 4 est transformé en un flottant.

Polymorphisme d'union/somme

La possibilité de construire des types représentants plusieurs types différents la résolution peut être statique/dynamique une partie est parfois laissée au programmeur.

En Ocaml

Les types somme et le mécanisme de filtrage. Ce mécanisme permet au programmeur d'identifier le type réel d'un objet (d'une instance de type somme par exemple).

```
type complex = C of float*float | R of float | I of float ;;
```

On a défini un type `complex` qui est :

- soit composé de deux flottants, ceux la sont construits avec le constructeur `C`
- soit composé d'un seul flottant construit avec le constructeur `R` (sémantique une partie réelle)
- soit composée d'un seul réel avec le constructeur `I` (ici juste une partie imaginaire)

```
let print c = match c with C (x, y) -> "vrai complex"
    | R(x) -> "complex réel"
    | I(x) -> "complex imaginaire" ;;

print complex->string = <fun>
```

En C

```
struct 2reels {
    float x;
    float y;
};

union complex {
    float f;
    2reels c;
};

struct nombre {
    complex c;
    enum type {C, R, I} T;
};

char *print (nombre n) {
    if (n.T ==C) {
        return "on attent un vrai complex de type 2reels";
    }
    else if (n.T == R) {
        return "on attend un complex reel";
    }
    else {
```

```
        return "on attend un complex imaginaire";
    }
}
```

Polymorphisme paramétrique

Polymorphisme paramétrique c'est le fait d'utiliser des variables qui représentent des types. Et ces variables sont instanciées statiquement.

On l'appelle dans certains langages : **généricité**. À priori c'est un polymorphisme *universel*. Un polymorphisme universel signifie que tout élément typable peut être instancié avec un nombre non borné de types.

Exemple : En `Ocaml`

```
let id x = x;;
    val id : 'a -> 'a = <fun> (* Variable représentant n'importe quel type *)
```

En `C++`

```
template <typename T> // on déclare le besoin de définir une fonction générique
T id(T x) {
    return x;
}
```

Un polymorphisme paramétrique implicite : le système de types s'occupe de la gestion automatique des variables de type. (Exemple : Ocaml)

Généralisation de la notion de système de types implicite. Sinon c'est explicite (i.e. on déclare les noms des types) (Exemples : C++, Java, …)

Le système de types va ajouter dans le code du programme une fonction appelée par exemple `id_chart`, qui remplace toute occurrence de `id(s)` où `s` est un `char[]`, par `id_chart`.

Le polymorphisme paramétrique contraint on peut typer les variables de types (?).

→ Cela signifie que l'on peut donner des propriétés que les valeurs de certaines variables de types doivent satisfaire.

Exemple : Java, ML (foncteurs).

NB : Le polymorphisme paramétrique en C++ n'est pas contraint.

Exemple de polymorphisme contraint en Java :

```
class C <T implements Runnable> {
    /* ~ */
}
```

Les valeurs possibles pour `T` sont exactement les classes qui implémentent `Runnable`.

Supposons ces 2 fonctions en C

```
int f() {
    /* code */
}

void f() {
    /* code */
}
```

Ces deux fonctions dans le même fichier produit une erreur de redéfinition → À priori le type de retour n'est pas pris en compte dans la définition des fonctions. Car idéalement `int f() {}` devrait représenter une fonction de type `void→int`. `void f() {}` une fonction de type `void→void`.

En C on peut appeler une fonction sans utiliser la valeur de retour. Cette ambiguïté n'est pas gérée par le compilateur.

En `Ocaml`

```
let s a b = a+b;;
    val s : int -> int -> int = <fun>
let s a b = a+.b;;
    val s : float -> float -> float = <fun>
s 5 10;;
Error: This expression has type int but an expression was expected of type
         float
(* La première définition n'existe plus dans le système s*)
```

Il n'a pas surchargé mais *redéfinit* la fonction `s`.

```
let f x y = (s x y);; (* on appelle la fonction s sur les entrées x et y *)
```

Ici le système de types si on permettait la redéfinition de s, il serait incapable de choisir parmi les deux car rien ne permet dans `(s x y)` d'inférer les types de `x` et `y`.

Syntaxe C++ (Différence avec C si besoin)

L'objectif est d'introduire à la fin les fonctions/classes virtuelles.

Le système de types est principalement statique. On peut avoir une version dynamique, mais à la demande.

- Affichage / Lecture : deux fonctions génériques : `cout` et `cin`, équivalent de `printf` et `scanf` sans formatage.

  ```
  #include <iostream> // on définit les fonctions entrée / sortie
  using namespace std; // package standard appelé namespace en C++, module
  // cout et cin s'appellent en réalité std::cout et std::cin
  // facilité syntaxique
  ```

```
int main() {
    int x;
    cin >> x; // saisie entier
    cout << "entier saisi : " << x << endl;
    char s[100];
    cin >> s;
    cout << "chaine saisie : " << s << endl;
    return 0;
}
```

- Pour les fichiers

```
#include <ofstream> // écrire dans un fichier
#include <ifstream> // lire ""
int main() {
    ifstream f("fic1"); // ouverture en lecture ...
    char c;
    f.get(c); // lire un caractère stocké dans c
    ofstream g("fic2"); // ouverture en écriture
    g.put(c); // on écrit le caractère c
    return 0;
}
```

- Un type booléen `bool`.
- Les mêmes autres types de bases qu'en C. Par exemple pour `int`, versions `short`, `long`, `signed`, `unsigned`.
- Les constantes `const`, sauf qu'en C++ on doit les initialiser. La raison : les constantes sont par défaut internes en C++ et externes en C.
- En C++ toute variable a une valeur par défaut.

```
int x; // par défaut c'est 0
```

- Les *cast* sont comme en C (physique).
- Les `enum`, `union` sont comme en C. En revanche dans une `union` on peut encapsuler des fonctions contrairement au C. Pour rappel :

```
int x; // déclaration
int x = 0; // déclaration + définition
```