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'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