Exemple de méthodes et outils pour la correction des programmes [TOR]

I. Correction d'un programme [TOR 6]

A. Spécification d'un problème

Définition 1 **Un programme** transforme automatiquement une donnée en entrée en un résultat en sortie.

Définition 2 [TOR def 6.1] **La spécification d'un problème** comporte deux parties : la description des entrées valides (respectant des **préconditions**) et la description du résultat (respectant des **postconditions**).

Example 3 **Trié un tableau d'entiers** a la spécification suivante :

- **Entrée**: un tableau d'entiers $T = [t_0, ..., t_n]$
- **Sortie**: permutation du tableau en entrée qui le trie.

Remarque 4 On distinguera la nature générale des données (souvent le type en OCaml par exemple) qui sont des **conditions de bonne formation** et les propriétés supplémentaires qu'elles doivent vérifier, i.e. les préconditions.

Exemple 5 Un tableau d'entiers **bien formé** sera par exemple un tableau contiguës d'entiers 32 bits en mémoire.

Remarque 6 La spécification d'un algorithme peut être **raffinée** en y ajoutant des contraintes plus fortes. Par exemple, on peut spécifier le traitement de deux données égales dans un algorithme de tri ou forcer une certaine complexité.

B. Correction partielle [TOR 6.1]

Définition 7 **Un programme est correct** (dit correction *partielle*) si toute sortie renvoyé depuis une entrée valide respecte les post-conditions.

Définition 8 [TOR def 6.2] **Un invariant de boucle,** étant donnés un algorithme, une boucle de cet algorithme et des entrées valides de l'algorithme, est une propriété *vérifiée* avant le premier tour de boucle et *préservée* par chaque tour de boucle.

Propriété 9 [TOR prop 6.1]

Étant donnés un algorithme, une boucle de celui-ci et un invariant sur cette boucle, alors pour toute entrée valide de l'algorithme l'invariant est vérifié à la fin de l'exécution de la boucle.

Exemple 10 [TOR prog 6.6]

Algorithme du tri insertion.

```
void insertion_sort(int a[], int n) {
    for (int i = 1; i < n; i++) {
        // invariant:
        //     a[0..i[ est trié
        int v = a[i];
        int j = i;
        while (j > 0 && a[j-1] > v) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = v;
    }
}
```

DEV

Exemple 11 **L'algorithme de Dijkstra** permet de calculer les plus courts chemins entre un sommet et tous les autres dans un graphe pondéré dont les arêtes sont de poids positif. Cet algorithme est correct et termine.

Remarque 12 Si le graphe étudié par l'algorithme de Dijkstra admet des poids négatifs, l'algorithme n'est pas **correct**.

C. Terminaison [TOR 6.2]

Définition 13 L'exécution d'un programme **termine** si elle produit un résultat en un temps fini pour toute entrée valide.

Définition 14 [TOR def 6.3] **Un variant,** étant donné une boucle d'un algorithme, est une fonction des variables de l'algorithme vers $\mathbb{N}$ qui décroît **strictement** à chaque tour de boucle.

Remarque 15 Pour un algorithme récursif, un variant est une même fonction qui décroît à chaque appel récursif.

Remarque 16 De façon plus générale, un variant peut être une fonction strictement décroissante des variables du programme vers n'importe quel ensemble E muni d'un **relation bien fondée.**

Définition 17 [TOR def 6.18] **Une relation $\leq$ est bien fondée** sur E s'il n'existe pas de suite infinie strictement décroissante pour $\leq$.

Implémentation 18 [TOR prog 6.1] **Recherche Dichotomique**

```c
int binary_search(int v, int a[], int n) {
  int lo, hi = n;
  while (lo < hi) { // lo <= hi serait incorrect
    int mid = lo + (hi - lo) / 2;
    if (a[mid] == v) return mid;
    if (v < a[mid]) { mid = mid; } else { lo = mid + 1; }
  }
  return -1;
}
```

Exemple 19 **La recherche dichotomique** possède le variant de valeur : `hi - lo` $\in \mathbb{N}$, la taille de l'intervalle de recherche. Cette quantité positive décroît bien strictement, car l'intervalle de recherche décroît en taille à chaque itération.

Définition 20 **Un programme est totalement correct** s'il est correct et qu'il termine.

II. Méthodes manuelles pour la correction [TOR 5]

A. Bonnes pratiques de programmation

Motivation 21 Plusieurs programmes écrits différemment peuvent avoir le même comportement. Cependant, le code doit être lu par des humains et se doit donc d'être un maximum « lisible ».

Remarque 22 **Le code idiomatique** suit une convention. Elle peut être définie par la communauté utilisant un même langage ou un groupe de développeurs collaborant sur un même projet.

Remarque 23 Il n'existe pas de « meilleure manière » absolue d'écrire un programme. Il faut éviter une forme de dogmatisme envers les idiomes.

Remarque 24 **L'immuabilité** est un idiome commun pour les langages fonctionnels qui facilite le raisonnement autour d'un programme et notamment la preuve de sa correction.

Conseils 25 **Du code « propre »** est un critère subjectif. On dira le plus souvent que du code documenté, factorisé, aéré, réutilisable et indenté sera propice à être lu plus simplement.

Méthode 26 **On peut documenter une fonction** à l'aide d'un commentaire placé avant sa définition. On y précise la spécification de la fonction (notamment préconditions, postconditions, cas d'erreur).

Méthode 27 **Séparer le code en fichiers** permet de regrouper les fonctionnalités thématiquement. Cela rend la navigation du code source plus simple.

Définition 28 [TOR 5.3] **La programmation défensive** vise à vérifier les préconditions d'une fonction avant d'en exécuter le code. Ceci permet d'assurer la validité des préconditions supposées par un programme.

Exemple 29 L'algorithme de Dijkstra pourrait faire un pré-calcul pour vérifier qu'un graphe n'a pas d'arêtes de poids négatifs.

Méthode 30 **Des assertions** peuvent être placées (`assert` en C ou OCaml) dans le code pour arrêter l'exécution en cas de précondition ou d'invariant non vérifié.

Remarque 31 On peut également renvoyer des valeurs d'erreurs ou lever des exceptions en cas de précondition ou d'invariant non-vérifié. Il est alors de la responsabilité de la fonction appelante de gérer les cas d'erreur.

Méthode 32 **Un débogueur** permet d'exécuter un code instruction par instruction en observant l'état du programme pour rechercher la cause d'un bogue.

Exemple 33 **GDB et LLDB** sont des débogueurs utilisés en C.

Remarque 34 En cas de comportement inattendu du code, déboguer son programme permet d'enquêter sur l'origine du problème.

Méthode 35 Les compilateurs renvoient des messages d'erreurs et avertissements à destination du développeur. On les activera en C avec les options `-Wall -Wextra -pedantic`.

B. Détection d'erreurs par les tests

Définition 36 **Un test unitaire** vérifie le comportement d'une unité « atomique » de code pour une entrée donnée (par exemple une unique fonction).

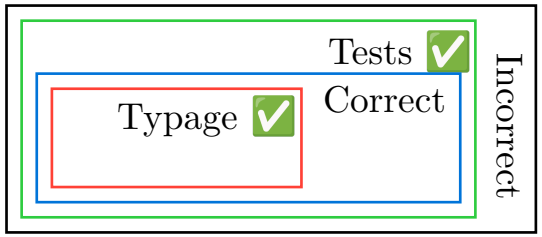
Définition 37 **Un test d'intégration** vérifie le bon comportement d'une combinaisons d'opérations (par exemple ouvrir un fichier, puis lire son contenu et le transformer en graphe).

III. Outils automatiques pour la correction

Théorème 38 **(Rice)** Toute propriété non trivial sur un programme est indécidable.

Conséquence 39 Automatiser la vérification de correction des programmes en général est indécidable.

Schéma 40 L'ensemble des programmes qui passent les tests sont une sur-approximation des problèmes correctes tandis que le typage sous-approxime.



A. Logique de Hoare [SEM 9.2]

Définition 41 **Un triplet de Hoare,** noté $\{P\}\ S\ \{Q\}$, est une affirmation avec P, Q des prédicats et S un programme. Cette formule signifie que dans l'état initial P est valide, si on exécute le programme S et qu'il termine alors, on arrive à un état final où Q est valide.

Exemple 42 **Triplet de Hoare de factoriel.**

$$\{x = n\}$$
$$y := 1; \text{while } \neg(x = 1) \text{ do } (y := x \times y; x := x - 1)$$
$$\{y = n! \wedge n > 0\}$$

Définition 43 **La logique de Hoare** utilisent des règles d'inférence permettant de prouver les triplets de Hoare.

Exemple 44 **L'outils WHY3** est un assistant de preuve de programme utilisant la logique de Hoare.

B. Typage [TOR 10.5.3] [TOR 3.1.3]

Citation 45 « A well typed program cannot go wrong », R. Miller.

Définition 46 **Le typage** en OCaml consiste à associer à chaque expression un type à la compilation. Ceci limite les programmes que l'on peut écrire et vise à **encoder dans le système de type** des conditions sur les valeurs des expressions.

Exemple 47 Ce code OCaml suivant ne compile pas :

```
let f (x : int) : int = 2 + x
let () = f "test"
```

Remarque 48 Les systèmes de type « communs » ne permettent pas de définir certaines préconditions. Par exemple, on ne peut pas en OCaml indiquer qu'une fonction n'accepte en entrée que des entiers de l'intervalle 0 à 10 de façon simple.

DEV

Méthode 49 **L'inférence de type** [TPL 22] évite au programmeur de typer chaque valeur. Le compilateur infère un type « le plus général » pour chaque valeur, et renvoie une erreur de compilation en cas de conflit.

Exemple 50 Le programme suivant typera `a`, `c` et `b` à `int`, et `f` : `int -> int -> int`.

```
let f a b = let c = a + b in a * c
```

C. Automatisation des tests

Définition 51 **La couverture d'un test** est l'ensemble des chemins dans le code parcourus lors de l'exécution d'un test.

Remarque 52 La couverture des tests peut être plus ou moins précise, allant jusqu'à vérifier que chaque instruction est bien exécutée au moins une fois.

Remarque 53 **Des outils de test de couverture** permettent de tester quelle partie du code est **couverte** par un jeu de test. La couverture d'un jeu de test est un critère de qualité simple.

Exemple 54 **GCOV** est un outils classique de test de couverture de code utilisé en C.

Définition 55 **Le Fuzzing** consiste à générer aléatoirement des entrées du programme et génére donc des test. On cherche ici à augmenter la couverture des tests existants.

Exemple de méthodes et outils pour la correction des programmes [TOR]

I. Correction d'un programme [TOR 6]

A. Spécification d'un problème

1 Def Un programme
2 Def [TOR def 6.1]La spécification d'un problème
3 Ex Trié un tableau d'entiers
4 Rem
5 Ex
6 Rem

B. Correction partielle [TOR 6.1]

7 Def Un programme est correct
8 Def [TOR def 6.2]Un invariant de boucle,
9 Prop [TOR prop 6.1]
10 Ex [TOR prog 6.6]
11 Ex L'algorithme de Dijkstra
12 Rem

C. Terminaison [TOR 6.2]

13 Def Terminaison
14 Def [TOR def 6.3]Un variant,
15 Rem
16 Rem
17 Def [TOR def 6.18] Une relation $\leq$ est bien fondée
18 Implem [TOR prog 6.1] Recherche Dichotomique
19 Ex La recherche dichotomique
20 Def Un programme est totalement correct

II. Méthodes manuelles pour la correction [TOR 5]

A. Bonnes pratiques de programmation

21 Motiv
22 Rem Le code idiomatique
23 Rem Code idiomatique $\neq$ dogmatique
24 Rem L'immuabilité
25 Conseils Du code « propre »
26 Métho On peut documenter une fonction
27 Métho Séparer le code en fichiers
28 Def [TOR 5.3] La programmation défensive
29 Ex
30 Métho Des assertions
31 Rem Propagation d'erreur
32 Métho Un débogueur

B. Détection d'erreurs par les tests

33 Ex GDB et LLDB
34 Rem
35 Métho
36 Def Un test unitaire
37 Def Un test d'intégration

III. Outils automatiques pour la correction

38 Thm (Rice)
39 Conséquence
40 Schéma

A. Logique de Hoare [SEM 9.2]

41 Def Un triplet de Hoare,
42 Ex Triplet de Hoare de factoriel.
43 Def La logique de Hoare
44 Ex L'outils WHY3

B. Typage [TOR 10.5.3] [TOR 3.1.3]

45 Citation
46 Def Le typage
47 Ex
48 Rem
49 Métho L'inférence de type [TPL 22]
50 Ex

C. Automatisation des tests [NAN]

51 Def La couverture d'un test
52 Rem Granularité couverture
53 Rem Des outils de test de couverture
54 Ex GCOV
55 Def Le Fuzzing

Remarque

En fonction des affinités (et développements choisis) on pourra développer plus en détails :

- la partie sur les tests
- la partie sur la logique de Hoare
- ajouter une partie sur le flot de contrôle et les tests liés à ça
- ajouter des choses que vérifie les compilos dans un programme

Attention relation $\neq$ ordre, une relation bien fondée peut suffire pour un invariant. On a ici parlé d'ordre comme le fait [TOR] .

A savoir

- comment passer d'un ordre $\leq$ à un ordre stricte $<$.
- bien savoir expliquer le triple de Hoare sur factoriel (connaitre l'arbre de dérivation)
- bien être au clair sur le théorème de Rice et les limitations de la partie 3.
- lien sur les systèmes de type interessant pour les questions.

Développements

- un développement sur la logique de Hoare peut être interessant.
- la preuve de la correction d'un algorithme fait un très bon développement (dikstra étant l'exemple canonique).
- développement sur une notion du typage est interessante.
- un développement sur les tests

Bibliographie

[TOR] T. Balabonski & S. Conchon & J. Filliâtre & K. Nguyen & L. Sartre, *MP2I MPI, Informatique Cours et exercices corrigés.*

[SEM] H. R. Nielson & F. Nielson, *Semantic with Applications An Appetizer.*

[TPL] B. Pierce, *Types and Programming Languages.*

Paul Adam & Aloïs Rautureau