Génie Logiciel Avancé

Léo Andrès

30 Janvier 2026

Évaluer la qualité d'une suite de tests

1. La couverture du code
    1. En OCaml : `bisect_ppx`
2. Représentations structurelles d'un programme
    1. Graphe d'appel
    2. Graphe de flot de contrôle
    3. Arbre d'exécution
3. Critères de couverture formels
    1. Définition des critères de couverture formels
    2. Gestion des critères de couverture par les labels
4. Test par mutation
    1. En OCaml : `mutaml`

La couverture du code

La couverture du code

Une fois qu'on a écrit des tests, on va mesurer la qualité de ceux-ci.

- quelle portion du code est effectivement testée ?
- définition de « portion du code » ?
- notion de couverture du code par les tests

Bisect_ppx

En OCaml, on peut utiliser `bisect_ppx`.

On commence par indiquer à `dune` les parties du code pour lesquels on veut mesurer la couverture :

```
(library
 ; ...
 (instrumentation
  (backend bisect_ppx))
)

(executable
 ; ...
 (instrumentation
  (backend bisect_ppx))
)
```

Bisect_ppx

Ajouter un champ `instrumentation` n'a aucun effet sur le programme tant qu'on n'indique pas à `dune` d'activer l'instrumentation ainsi :

```
$ BISECT_FILE=$(pwd)/bisect dune runtest --force --instrument-with bisect_ppx
```

Va alors automatiquement modifier le code (l'instrumenter) pour ajouter des instructions qui vont compter le nombre d'exécutions des différentes parties.

Bisect_ppx

On peut alors générer un rapport ainsi :

```
$ bisect-ppx-report summary
Coverage: 9416/12003 (78.45%)
```

Bisect_ppx

On peut générer le rapport par fichier :

```
$ bisect-ppx-report summary --per-file
 93.51 %    245/262     src/bin/owi.ml
 87.02 %    114/131     src/cmd/cmd_c.ml
 73.60 %    131/178     src/cmd/cmd_call_graph.ml
 85.19 %     69/81      src/cmd/cmd_cfg.ml
 87.63 %     85/97      src/cmd/cmd_cpp.ml
 95.00 %     19/20      src/cmd/cmd_fmt.ml
 76.92 %     10/13      src/cmd/cmd_instrument_label.ml
 61.41 %    113/184     src/cmd/cmd_iso.ml
 54.19 %    110/203     src/cmd/cmd_replay.ml
 77.78 %      7/9       src/cmd/cmd_run.ml
 86.96 %     40/46      src/cmd/cmd_rust.ml
# ... truncated for readability
 78.45 %   9416/12003   Project coverage
```

Bisect_ppx

On peut aussi générer une version HTML avec des détails sur chaque portion du code :

```
$ bisect-ppx-report html -o _coverage
```

Le rapport produit est alors inspectable via notre navigateur :

```
$ xdg-open _coverage/index.html
```

→ démo

Représentations structurelles d'un programme

Représentations structurelles d'un programme

Des façons de représenter un programme ou un ensemble d'exécutions de celui-ci.

Permet de mieux comprendre et décrire diverses analyses.

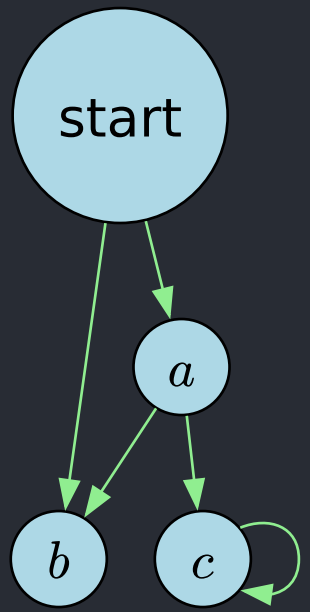
1. Graphe d'appel
2. Graphe de flot de contrôle
3. Arbre d'exécution

Graphe d'appel

- une représentation statique du programme
- un graphe
- chaque sommet représente une fonction
- les arcs sont orientés
- un arc de f vers g indique qu'il y a un appel de f vers g

Graphe d'appel

```
(module

  (func $start
    i32.const 0
    (if
      (then call $a)
      (else call $b)))

  (func $a
    (block
      call $b)
    call $c)

  (func $b)

  (func $c
    call $c))
```



Graphe d'appel

On ne peut pas calculer le graphe d'appel exact de n'importe quel programme : problème indécidable.

Lié aux appels indirects :

- présents dans la majorité des langages de programmation
- se dit d'un appel où on n'utilise pas directement le nom d'une fonction (`call $f`) mais où on passe par exemple par une référence, une variable ou un objet (`call_indirect $function_table (type $t) (i32.const 42)`)

Graphe d'appel

```
void hello(void) {
  printf("Hello\n"); }

void goodbye(void) {
  printf("Goodbye\n"); }

void never(void) {
  printf("Never\n"); }

void (*funcs[])(void) = { hello,
goodbye, never };

void main(void) {
  fptr = funcs[rand() % 2];
  fptr(); }
```

- `funcs` peut être modifié dynamiquement
- on ne sait pas que ce que peut renvoyer `rand`
- savoir que `% 2` ne permet d'accéder qu'au deux premiers élément n'est pas forcément évident

Graphe d'appel

À cause de cette indécidabilité, deux façons de calculer les graphes d'appels :

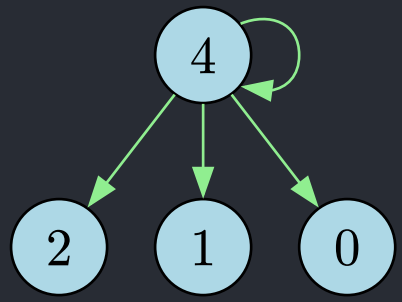
1. Par sur-approximation : on accepte d'acoir des arcs en trop par rapport à ce qui est possible en pratique, mais il n'en manque aucun.
2. Par sous-approximation : on accepte d'avoir des arcs en moins par rapport à ce qui est possible en pratique, mais ils sont tous possibles.

Les deux ont leur utilité :

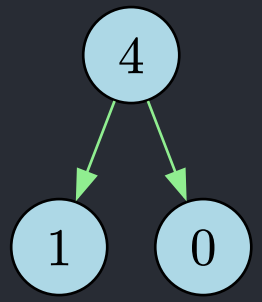
1. optimiseur qui veut remplacer les appels indirects par des appels direct quand il n'y a qu'une seule valeur possible
2. un outil de recherche de bug qui ne veut pas lever de fausse alarme

Graphe d'appel

```
(module
  (import "env" "unknown" (func $unknown
(result i32)))
  (table $mytable 2 funcref) ;; a function
table
  (type $t0 (func))

  (func $hello)
  (func $goodbye)
  (func $never (param i32))

  (func $start
    call $hello
    call $unknown
    call_indirect $mytable (type $t0))

  (start $start))
```



sur-approx.



sous-approx.

Graphe de flot de contrôle

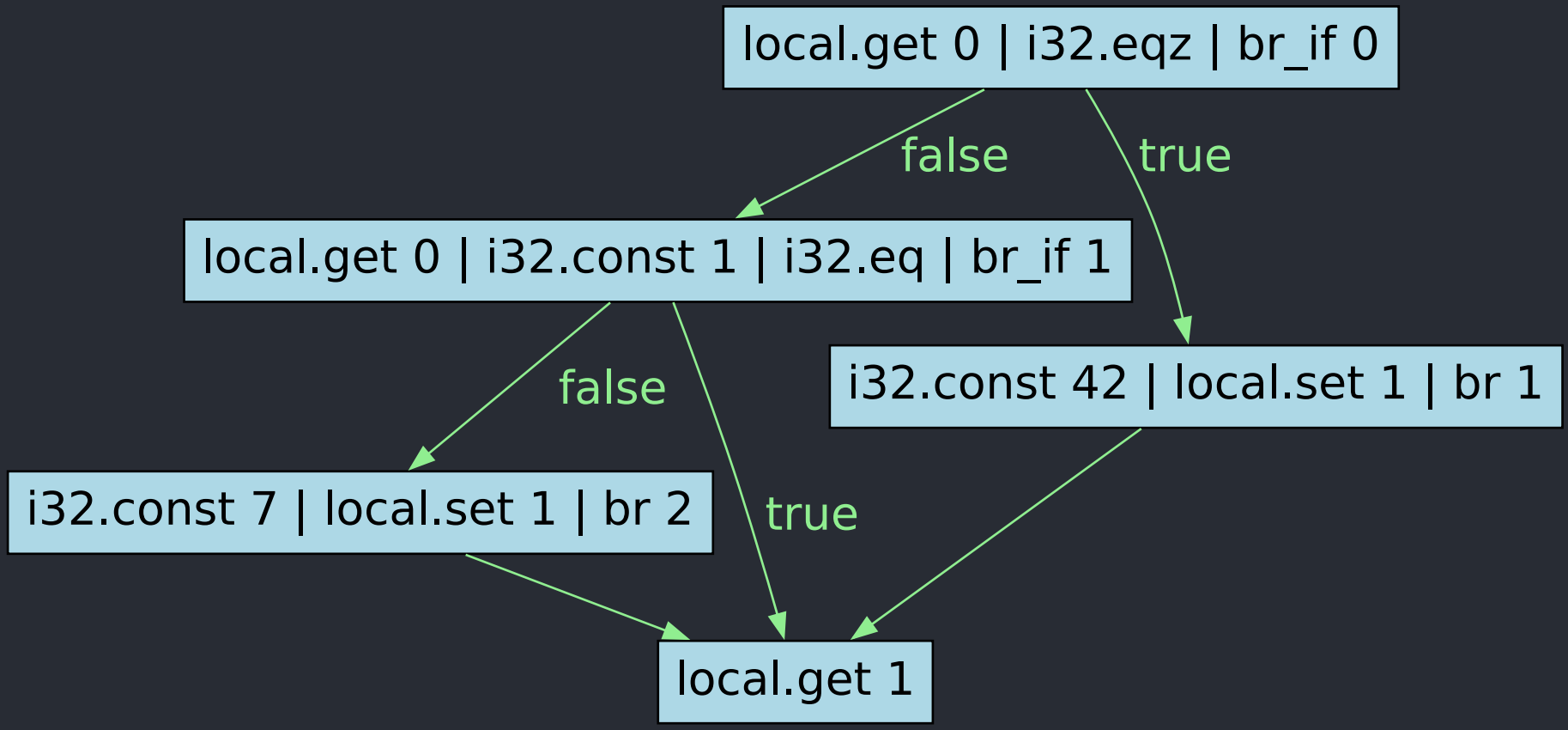
Une autre représentation statique du programme.

- existe sur plusieurs fonctions à la fois, ici, une seule fonction à la fois
- représente les différents branchements (ou « sauts ») possibles
- toujours un graphe
- les sommets sont des expressions (souvent une suite d'instructions) sans branchement
- les arcs sont des branchements

Graphe de flot de contrôle

```
(module
  (func $f (param i32) (result i32) (local i32)
    (block
      (block
        (block
          (block
            (i32.eqz (local.get 0))
            br_if 0
            (i32.eq (i32.const 1) (local.get 0))
            br_if 1
            (local.set 1 (i32.const 7))
            br 2)
          (local.set 1 (i32.const 42))
          br 1)
        local.set 1 (i32.const 99))
      local.get 1))
```

Graphe de flot de contrôle



Graphe de flot de contrôle

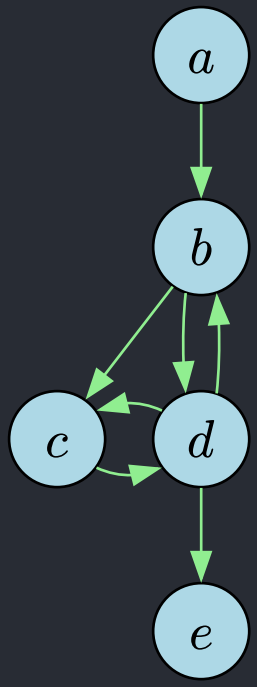
Le flot de contrôle d'un programme peut être structuré ou non structuré. Il est struturé si son graphe de flot de contrôle est réductible. Ce qui est le cas si :

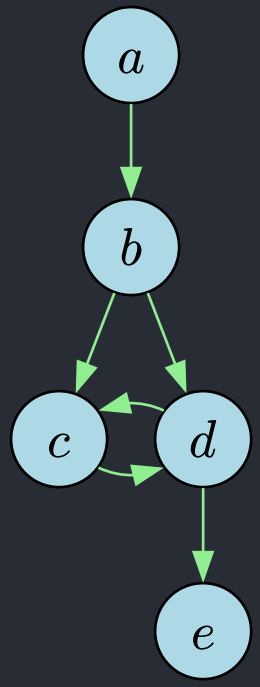
- après avoir retiré tous les arcs de retours on obtien un graphe acyclique et dont tous les sommets sont accessibles depuis l'entrée du graphe

Un arc de B vers A est un arc de retour si et seulement si A domine B.

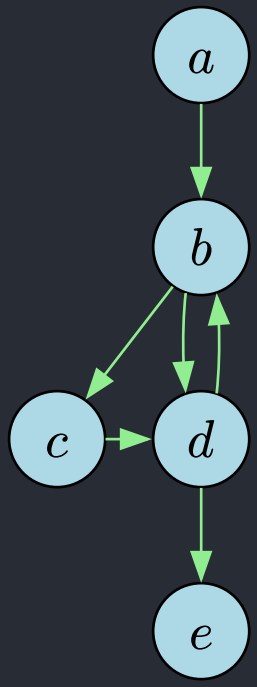
A domine B si et seulement si tous les chemins depuis l'entrée du graphe vers B passent par A. I.e. si on doit forcément passer par A pour atteindre B.

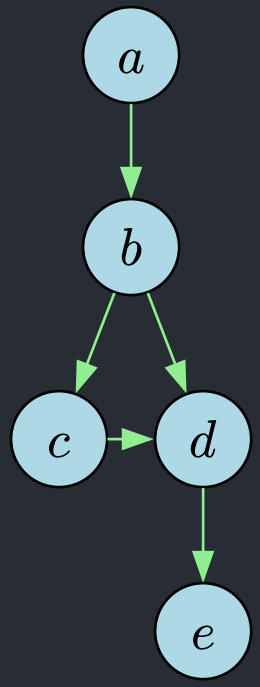
Graphe de flot de contrôle





Graphe de flot de contrôle



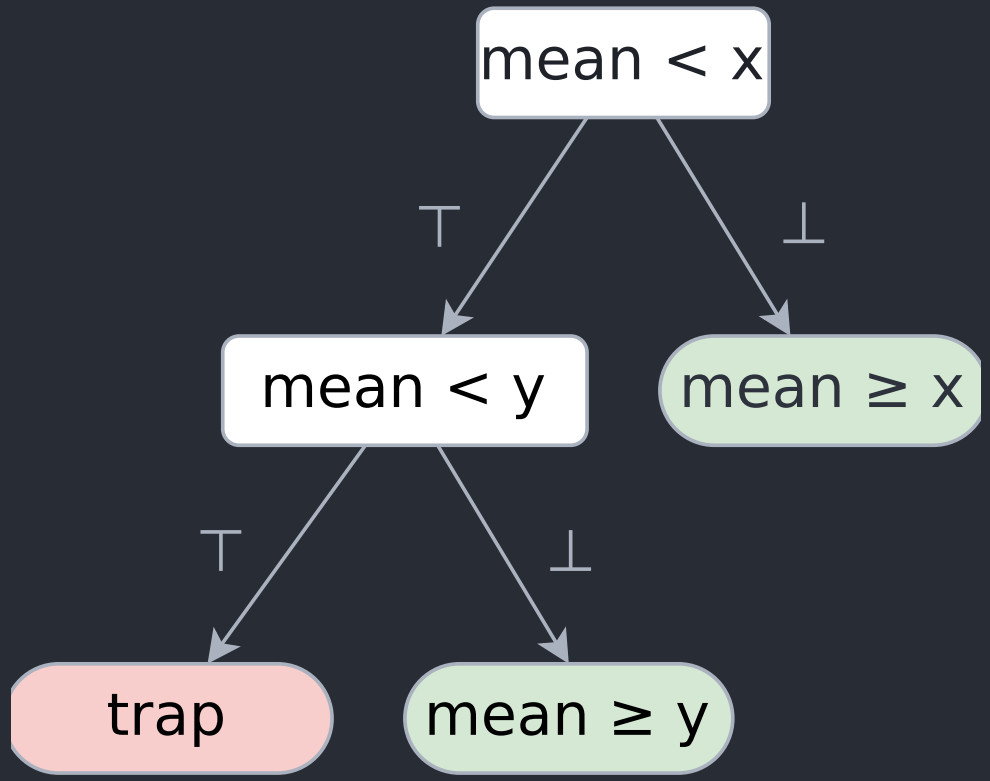


Arbre d'exécution

- représente toutes les exécutions possibles d'un programme en fonction de ses valeurs d'entrées
- on ne va s'intéresser ici qu'aux fonctions
- similaire au graphe de flot de contrôle
- mais potentiellement infini (le programme peut ne pas terminer)
- pas de cycle, c'est donc un arbre et pas un graphe (on a « déroulé les boucles »)

Arbre d'exécution

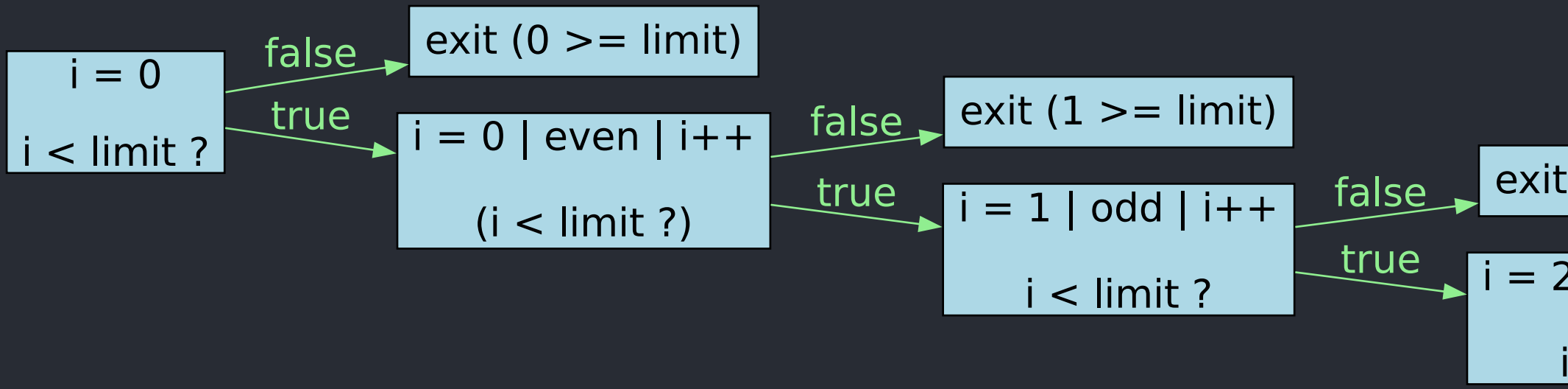
```
unsigned int mean(unsigned int x,
                  unsigned int y) {

  unsigned int mean = (x + y) / 2;
  if (mean < x) {
    if (mean < y) {
      assert(false);
    }
  }
  return mean;
}
```



Arbre d'exécution

```c
#include <stdlib.h>

void f(int limit) {
  int i = 0;

  while (i < limit) {
    if (i % 2 == 0) {
      // even case
    } else {
      // odd case
    }
    i++;
  }
}
```

Arbre d'exécution



On ne s'intéresse qu'aux cas possibles en pratique. Par exemple, ici, on ne représente pas le cas où `i = 0` et où on aurait exécuté `odd`.

Critères de couverture formels

Critères de couverture formels

On va définir plusieurs façons de mesurer la couverture du code par les tests.

On appelle ces différentes mesures des critères de couverture.

On verra ensuite comment tous les encoder en un mécanisme unifié (les labels).

Enfin, on verra comment instrumenter le code automatiquement pour mesurer la couverture selon n'importe quel ensemble de critères.

Couverture des fonctions

- appelé FC (function coverage)
- mesure le nombre de fonctions exécutées par les tests
- sur le graphe d'appel, cela correspond à vérifier que chaque sommet a été exécuté au moins une fois
- si notre programme a deux fonctions f et g et que nos tests appellent seulement f (et que f n'appelle pas g), alors on aura 50% de couverture selon le critère FC

Couverture des appels de fonction

- appelé FCC (function call coverage)
- mesure le nombre d'appels de fonctions exécutés par les tests
- sur le graphe d'appel, cela correspond à vérifier que chaque arc du graphe a été emprunté au moins une fois
- généralement, le cas des appels indirects est simplifié et on vérifie seulement si on est passé par l'appel indirect une fois peu importe la fonction qui était la cible de l'appel

Couverture des instructions

- appelé `Stmt` (statement coverage)
- revient à dire que chaque instruction a été exécuté une fois
- équivaut à peu près à dire que chaque sommet du graphe de flot de contrôle a été exécuté (si on exécute la première, les autres devraient l'être, sauf en cas d'erreur, parce qu'il n'y a plus de branchements)

Couverture des décisions

- appelé DC (decision coverage)
- cherche à mesurer si pour chaque branchement conditionnel on a pris au moins une fois les deux décisions possibles (prendre le branchement ou bien ne pas le prendre)

```
if (p && q) {
  // ...
}
```

1. `p && q`
2. `!(p && q)`

Si on exécute seulement un des deux, on a 50% de couverture.

Correspond à dire que chaque arc du flot de contrôle a été emprunté.

Couverture des conditions

- appelé `cc` (condition coverage)
- vérifie si pour chaque proposition atomique d'une condition, on a essayé toutes les valeurs

```
if (p && q) {
  // ...
}
```

1. `p`
2. `!p`
3. `q`
4. `!q`

Couverture des conditions

Attention, `DC` n'implique pas `CC` et l'inverse non plus. Sur l'exemple `p && q`, on va avoir :

1. `p == true ; q == false` et `p == false ; q == true` qui donnent 100% sur `CC` mais seulement 50% sur `DC`.
2. `p == true; q == false` et `p == true ; q == false` qui donnent 100% sur `DC` mais seulement 75% sur `CC`

En Wasm, ils sont équivalents parce que tous les branchements se font sur des valeurs atomiques (pas de sous-expressions).

Couverture des conditions

`CC` est assez particulier parce que deux programme équivalents sémantiquement peuvent avoir des scores différents sur des entrées identiques. Par exemple `x == true ; y == true` et `== true ; y == false` donnent 100% sur `g` mais 75% sur `f`.

```
void f(int x, int y) {
  if (x && y) { // ...
  }
}

void g(int x, int y) {
  int condition = x && y;
  if (condition) { // ...
  }
}
```

Couverture des conditions

cc est donc une forme de critère sur les arcs du flot de contrôle mais où l'on utiliserait un graphe de flot de contrôle différent de celui usuel et qui aurait été réécrit pour transformer toutes les conditions en des conditions atomiques. (c'est à peu près ça mais pas vraiment)

Couverture des conditions multiples

- appelé `MCC` (multiple condition coverage)
- cheche à mesurer si, pour chaque condition, on a exécuté l'ensemble des combinaisons possible pour toutes les valeurs des conditions atomiques possibles

Si on repart sur `if (p && q) { .. }`, pour obtenir 100% de couverture, il faut l'exécuter avec :

1. `p == false; q == false`,
2. `p == false; q == true`,
3. `p == true; q == false`,
4. `p == true; q == true`.

Couverture des conditions multiples

`MCC` implique `CC`.

Il implique aussi `DC` sauf dans les cas où la condition globale est toujours vraie ou toujours fausse :

```
if (x && !x) {
  ...
}
```

On peut obtenir 100% sur `MCC` avec `x == true` et `x == false`, mais on obtiendra jamais plus de 50% sur `DC`.

Autres critères de couverture

Il en existe plein d'autres : `MC/DC`, `NCC`, `GACC`, `GICC`, `LIMIT`, `ELO`, `SLO`, `DUC`, `ADC`, `AUC`, `WM`, `IOB`, `CB`, `BC`, `IDP`.

Dans l'aéronautique, on impose une couverture de 100% selon le critère `MC/DC`.

On va maintenant voir comment encoder et générer des labels pour rendre ces mesures automatiques.

Gestion des critères de couverture par les labels

L'idée consiste à essayer de représenter tout ces critères de la même façon dans notre programme. Cette notion s'appelle les labels. Les labels sont simplement des appels à une fonction `label` qu'on a inséré dans le programme, et qui vont vérifier, pendant l'exécution des tests, si on couvre tout ce qu'on est censés couvrir pour un critère donné ou un ensemble de critères.

Couverture des fonctions

Pour FC :

On transforme :

```
int f(void) {

  // ...
}

int g(void) {

  // ...
}
```

En :

```
int f(void) {
  label("FC", 1, true);
  // ...
}

int g(void) {
  label("FC", 2, true);
  // ...
}
```

Couverture des appels de fonction

Pour FCC :

On transforme :

```
int f(void) { // ...
}
int g(void) {
  if (...) {
    // ...
    f();
  } else {
    // ...
    f();
  }

  f();}
```

En :

```
int f(void) { // ...
}
int g(void) {
  if (...) {
    // ...
    label("FCC", 1, true); f();
  } else {
    // ...
    label("FCC", 2, true); f();
  }
  label("FCC", 3, true); f(); }
```

Couverture des décisions

Pour `DC` :

On transforme :

```
if (p && q) {
  // ...
}
```

En :

```
label("DC", 1, p && q);
label("DC", 2, !(p && q));
if (p && q) { // ...
}
```

Ou :

```
if (p && q) {
  label("DC", 1, true); // ...
} else {
  label("DC", 2, true);
}
```

Couverture des conditions

Pour `cc`:

On transforme :

```
if (p && q) {
  // ...
}
```

En :

```
label("CC", 1, p);
label("CC", 2, !p);
label("CC", 3, q);
label("CC", 4, !q);
if (p && q) {
  // ...
}
```

Couverture des conditions multiples

Pour MCC:

On transforme :

```
if (p && q) {
  // ...
}
```

En :

```
label("MCC", 1, !p && !q);
label("MCC", 2, !p && q);
label("MCC", 3, p && !q);
label("MCC", 4, p && q);
if (p && q) {
  // ...
}
```

Outils pour la gestion de labels

- on ne va pas annoter les programmes à la main pour tous les labels
- on va utiliser des outils qui vont instrumenter notre programme
- on leur donne un programme et un ou plusieurs critères
- ils génèrent un nouveau programme équivalent à l'original et où les bons labels ont été ajoutés

LAnnotate

Pour le langage C, l'outil `LAnnotate` de Frama-C permet de faire cela et supporte de très nombreux critères et plein d'options de génération. Il s'utilise ainsi :

```
$ frama-c -lannot=CC,DC file.c
```

On aura alors un fichier `file_labels.c` produit, lequel contiendra les labels pour les critères demandés (ici, `CC` et `DC`). C'est ce fichier qui doit être utilisé lorsque l'on teste notre programme afin de mesurer la couverture.

owi instrument label

Pour Wasm, Owi supporte les critères les plus courants et s’utilise ainsi :

```
$ owi instrument label --criteria=dc file.wat
```

Test par mutation

Test par mutation

Une technique pour mesurer l'efficacité d'une suite de tests. L'idée est simple :

1. On ajoute des bugs dans le programme.
2. On lance la suite de tests.
3. Si la suite de tests se met à échouer, elle a bien trouvé le bug, sinon, c'est qu'on a trouvé un cas qui n'était pas testé.

Test par mutation

Pour ajouter des bugs, on va simplement effectuer des petites modifications dans le programme. Par exemple, on peut :

- échanger des opérateurs booléens comme `&&` et `||` ;
- échanger des opérateurs entiers comme `+`, `-`, `*`.
- changer la valeurs de certaines constantes.

D'autres idées ? Il y a énormément de possibilités…

Test par mutation

Chaque modification du programme est appelé une mutation et chaque programme contenant des mutations est appelé un mutant. Une fois qu'on sait les générer, il y a deux approches possibles :

1. Essayer différents mutants jusqu'à trouver un bug, on peut alors écrire un nouveau test à partir de celui-ci.
2. Effectuer l'opération un grand nombre de fois et calculer un score de mutation qui correspond au pourcentage de fois où notre suite de test a été capable de détecter un mutant.

Mutaml

En OCaml, on peut faire du test par mutation avec un outil appelé `mutaml`. On commence par indique à `dune` qu'on souhaite instrumenter le projet avec `mutaml` :

```
(executable
  (public_name mytool)
  (instrumentation (backend mutaml)))

(library
  (public_name mylibrary)
  (instrumentation (backend mutaml)))
```

Mutaml

On peut alors compiler notre projet en activant l'instrumentation ainsi :

```
$ dune build --instrument-with mutaml
```

```
      ppx lib.pp.ml
Running mutaml instrumentation on "lib.ml"
Randomness seed: 810959211   Mutation rate: 50   GADTs enabled: false
Created 9 mutations of lib.ml
Writing mutation info to lib.muts
```

Mutaml

Pour un test donné, par exemple `test/mytests.ml`, on peut lancer le processus de test par mutation ainsi :

```
$ mutaml-runner _build/default/test/mytests.exe
```

```
read mut file lib.muts
Testing mutant lib:0 ... failed
Testing mutant lib:1 ... failed
Testing mutant lib:2 ... failed
Testing mutant lib:3 ... failed
Testing mutant lib:4 ... failed
Testing mutant lib:5 ... passed
Testing mutant lib:6 ... failed
Testing mutant lib:7 ... failed
Testing mutant lib:8 ... failed
Writing report data to mutaml-report.json
```

Mutaml

Enfin, on peut générer un rapport avec la commande `$ mutaml-report` :

```
Attempting to read from mutaml-report.json...

Mutaml report summary:
----------------------

 target                        #mutations      #failed      #timeouts      #passed
 ------------------------------------------------------------------------------------
 lib.ml                                 9      88.9%    8     0.0%    0    11.1%    1
 ====================================================================================

Mutation programs passing the test suite:
-----------------------------------------

Mutation "lib.ml-mutant5" passed (see "_mutations/lib.ml-mutant5.output"):

--- lib.ml
+++ lib.ml-mutant5
@@ -11,7 +11,7 @@
 (* A Monte Carlo simulation computing a pi-approximation *)
 let pi total =
   let rec loop n inside =
-    if n = 0 then
+    if n = 1 then
       4. *. (float_of_int inside /. float_of_int total)
     else
       let x = 1.0 -. Random.float 2.0 in

------------------------------------------------------------------------
```

Questions

1. Le cours de la semaine passée ?
2. Le cours de cette semaine ?
3. Le projet ?
4. OCaml ?

Bonus