Preuves et correction de programmes

I Principes de la preuve de programme

Def 1 (Correction) : La correction est une propriété mathématique assurant qu'un programme respecte certaines propriétés

Def 2 (Spécification) : La spécification d'un programme est la donnée de l'ensemble des propriétés devant être vérifiée lors de la correction du programme.

Exemple : En python, la fonction math.sqrt(x) (appelée avec x réel) :
- renvoie $\sqrt{x}$ si $x \geq 0$
- lève l'exception ValueError sinon.

Def 3 (validation) : La validation est un processus qui assure la correction d'un programme sur toutes les entrées possibles. Elle s'effectue au moyen d'analyses statiques.

Def 4 (vérification) : La vérification est un processus qui assure que la spécification d'un programme est vérifiée sur un nombre fini d'entrées. Elle s'effectue au moyens de tests.

⚠ La vérification ne permet souvent pas de prouver la correction d'un programme.

IA. Types de propriétés, théorème de Rice.

Def 5 (propriété de sureté) : Une propriété de sureté décrit le fait qu'un programme n'entre pas dans un état jugé mauvais

Exemple : Le programme n'effectue pas de division par 0.

Def 6 (propriété de vivacité) : Une propriété de vivacité décrit le fait que le programme finira par atteindre un état jugé bon.

Exemple : Le programme calcule la factorielle de n et la stocke dans la variable res.

Def 7 (terminaison) : Un programme termine lorsqu'il effectue un nombre fini d'étapes avant de s'arrêter.

Note. La terminaison est une propriété de vivacité.

Def 8 (correction partielle) : On sépare généralement la preuve de la terminaison des autres propriétés de la spécification d'un programme. Sans la preuve de la terminaison, on parle de correction partielle. Avec la terminaison, on parle de correction totale.

Théorème 1 (Rice) : Toute propriété sémantique non triviale d'un programme est indécidable.

Exemples : Il n'y a pas d'algorithme prenant un programme p et un argument x qui permette de déterminer les propriétés suivantes :
- p termine sur l'entrée x
- p renvoie la valeur x^2 sur l'entrée x.

IB Modèle de programme

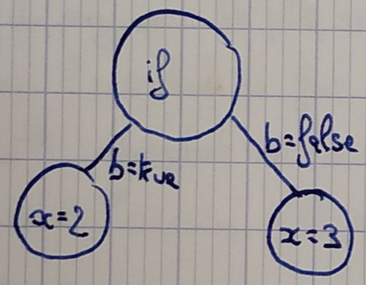
Les programmes et propriétés que l'on veut prouver sont parfois informels ou different selon le langage utilisé, il est donc utile de les abstraires pour construire des outils et algorithmes de preuve de programme.

Def 9 (modèle de programme): un modèle de programme est une abstraction du programme aidant à prouver des propriétés sur le programme

Exemple: On modélise le programme python suivant par son arbre de syntaxe abstraite:

```
if b:
    x=2
else:
    x=3
```

modélisation ⟶



Def 10 (formalisation): la formalisation de propriétés est une abstraction des propriétés de la spécification en formules mathématiques.

Exemple: La propriété "la factorielle de n est calculée et stockée dans la variable res" est abstraite par la formule "res = n!"

Les preuves de programme s'effectuent entre le modèle du programme et la formalisation de la spécification.

II Outils mathématiques utiles à la preuve de programmes.

IIA Induction et ordre bien fondé (programmes récursifs)

On utilisera comme exemple la fonction suivante (en OCaml):

```
let fact n = match n with
  | 0 -> 1
  | _ -> n x (fact (n-1))
```

Def 11 (ordre bien fondé): L'ordre $\leq$ est bien fondé sur l'ensemble E s'il n'existe pas de suite infinie décroissante pour $\leq$ dans E. $(E, \leq)$ est alors un ensemble bien fondé.

Théorème 2 (terminaison bien fondée): Soit $f: E \to X$ une fonction récursive définie sur $(E, \leq)$ bien fondée respectant:

- Pour tout $e \in E$, le calcul de $f(e)$ effectue un nombre fini d'appels à des $f(e')$, avec toujours $e' < e$
- Les autres étapes de calcul terminent

Alors le calcul de $f(e)$ termine pour tout $e \in E$.

Exemple: Ici, $(E, \leq) = (\mathbb{N}, \leq)$ et le calcul de fact n fait un unique appel à fact (n-1) (pour $n \geq 1$, et 0 sinon), donc le calcul de fact n termine pour tout $n \geq 0$.

Théorème 3 (correction par principe d'induction): Soit P un propriété sur $(E, \leq)$ bien fondée telle que

$$\forall e \in E, \left(\forall e' < e, P(e')\right) \Rightarrow P(e)$$

Alors $P(e)$ est vraie pour tout $e \in E$.

Exemple: On note $P(n)$ = "fact n = n!". Soit $n \in \mathbb{N}$

- si $n = 0$, alors fact 0 = 1 = 0! et on a $P(0)$
- si $n > 0$, et que $\forall n' < n$, $P(n')$, alors en particulier, on a $P(n-1)$, i.e. fact (n-1) = (n-1)!

  Ainsi, fact n = n x (fact (n-1)) = n x (n-1)! = n!

  Donc on a $P(n)$, et le théorème assure $P(k)$ $\forall k \in \mathbb{N}$

IIB Variants et Invariants (programmes itératifs)

On considérera l'exemple suivant (fonction en C):

```
int fact(int n){
    int res = 1;
    while (n!=0){
        res = res x n; n = n-1;
    } return res; }
```

Théorème 4 (terminaison par variant): Soit $v \in \mathbb{N}$ une quantité (dépendant des variables d'un programme) décroissant strictement au cours de l'exécution d'un programme. Alors le programme termine.

Exemple: On note v la valeur de n dans la fonction fact. On peut prouver par récurrence que $v \in \mathbb{N}$ tout au long du programme. De plus, v décroit strictement à chaque tour de boucle. Donc l'exécution de fact termine.

Def 12 (invariant): Un invariant est une quantité dépendant des variables d'un programme qui reste constante au cours de l'exécution. Elle est utilisée pour prouver la correction d'un programme.

Exemple: On note $i = res \times n!$

Par récurrence, on prouve que i est constant durant l'exécution de fact (R). Comme à la fin de la boucle, on a $n=0$, on en déduit que res contient $i/0! = i$.

Or, au début du programme, $i = k!$, ce qui prouve la correction partielle

II C Triplets et logique de Hoare

Def 13 (Pre (Post) condition): Une pré(post) condition est une propriété vérifiée sur les variables d'un programme avant (après) l'exécution d'une partie ou de tout le programme

Def 14 (Triplet de Hoare): Un triplet de Hoare est la donnée d'une précondition P, d'un fragment de programme C et d'une post-condition Q. Il est noté $\{P\}\ C\ \{Q\}$

Def 15: (Logique de Hoare): La logique de Hoare définit des règles permettant de prouver un triplet de Hoare. Les règles sont les suivantes

$$\frac{}{\{Q[e/v]\}\ v := e\ \{Q\}}\ (affP) \qquad \frac{\{P\}\ C_1\ \{Q\}\quad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1 ; C_2\ \{R\}}\ (seq) \qquad \frac{P \Rightarrow P' \quad \{P'\}\ C\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}}\ (weak)$$

$$\frac{\{E = true \wedge P\}\ C_1\ \{Q\}\quad \{E = false \wedge P\}\ C_2\ \{Q\}}{\{P\}\ \text{if } E \text{ then } C_1 \text{ else } C_2\ \{Q\}}\ (if)$$

$$\frac{\{E = true \wedge I \wedge V = z\}\ C\ \{I \wedge V < z\} \quad I \Rightarrow V \geq 0}{\{I\}\ \text{while } E \text{ do } C \text{ done } \{E = false \wedge I\}}\ (while)$$

Théorème 5 (correction de la logique de Hoare): La logique de Hoare est correcte, i.e. si $\{P\}\ C\ \{Q\}$ est prouvable, alors l'exécution du programme C avec les préconditions P vérifie les postconditions Q.

III Algorithmes de preuve de programme

III A. Typage

Def 16: Le typage associe à chaque variable du programme un type et vérifie que les opérations effectuées respectent bien les types. Le typage n'a pas toujours besoin d'être explicité pour être validé car il existe des algorithmes d'inférence de type.

Exemple: En Ocaml, `let s = "Bonjour" in s + 1` ne compile pas car la fonction + a le type int → int → int et s le type string (inféré à la compilation).

III B. Interprétation abstraite

Def 17: Un interpréteur abstrait est un algorithme qui approxime l'exécution d'un algorithme sur un domaine abstrait, choisi pour que l'approximation termine toujours et que les propriétés vérifiées soient vraies pour le programme original.

Devoir 2 Implémentation d'un interpréteur abstrait pour un mini-langage itératif.

III C. Assistants de preuve basé sur la logique

Def 18: Un assistant de preuve infère à partir des règles d'une logique un arbre de preuve pour une propriété. Une aide humaine est souvent nécessaire pour certaines

Exemple: Why3 est basé sur la logique de Hoare et Coq sur la logique intuitionniste. / règles