Projet

Logiciels sûrs

—Master Informatique, 1ière année, 2ème semestre, 2021-2022*—*

Un interpréteur pour un langage impératif et un prouveur en logique de Hoare

Les objectifs de ce projet sont de programmer un interpréteur pour un langage impératif simple (on sait que toutes les boucles terminent) et un prouveur (très simple) en logique de Hoare pour ce langage impératif.
Le projet est à réaliser en trinôme ou binôme (inscription sur UPdago obligatoire). Il est à rendre pour le 25 **mars** 2022 **sur Updago.**
Le format attendu pour le rendu du projet est un code correctement commenté muni d'exemples d'exécution pour chaque fonction. Aucun fichier pdf ne sera lu et pris en compte.
Le code doit être contenu dans une archive et débarrassé de tous les artéfacts introduits par les IDE utilisés. L'archive doit contenir un fichier Readme et doit être organisée de façon simple, claire et raisonnable.
Vous avez le choix de programmer en Java, C++ ou OCaml. **Je vous conseille très fortement d'utiliser OCaml qui est le langage le plus largement adapté à la programmation qui vous est demandée.** [a]

a. N'oubliez pas le proverbe « Quand on n'a qu'un marteau, tous les problèmes ressemblent à des clous ». Les langages de programmation sont vos outils, utilisez l'outil approprié au problème à résoudre et ne faites pas l'inverse en tordant le problème à résoudre afin de l'adapter au seul outil que vous connaissez.

---

1 Un interpréteur pour un langage impératif

Afin de simplifier le sujet, **nous ne travaillerons qu'avec la syntaxe abstraite** du langage que nous définissons.

Cette syntaxe est définie en trois parties, d'abord les expressions arithmétiques, puis les expressions booléennes et enfin les commandes du langage.

La syntaxe abstraite est la structure de données qui permet de représenter et de manipuler les expressions et commandes du langage en machine. Cette syntaxe abstraite est une structure arborescente dont les noeuds contiennent les « opérateurs » du langage (opérations arithmétiques, connecteurs logiques ou commandes) et les feuilles sont soit des variables (arithmétiques) ou des constantes (arithmétiques ou logiques).

Les noms des variables (i.e. les identificateurs) sont de type chaîne de caractères (i.e. du type `string` en OCaml).

1.1 Les expressions arithmétiques

1.1.1 Syntaxe abstraite

Dans cette partie la syntaxe abstraite des expressions arithmétiques est définie. En plus des constantes et des variables qui prennent toutes les deux des valeurs entières, les opérations demandées sont l'addition, la soustraction et la multiplication.

▸ **Question 1.** *Définissez un type* `aexp` *qui donne la syntaxe abstraite des expressions arithmétiques du notre langage.*

▸ **Question 2.** *À l'aide du type* `aexp` *écrivez les expressions suivantes :*

- 2
- $2 + 3, 2 - 5, 3 * 6$
- $2 + x, 4 * y, 3 * x * x, 5 * x + 7 * y, 6 * x + 5 * y * x$

▸ **Question 3.**

1. *Dans la question précédente, vous avez sans doute remarqué que la lecture de la syntaxe abstraite n'est très pas facile. Afin de rendre cette lecture moins fastidieuse, écrivez une fonction* `aexp_to_string` *qui transforme une expression arithmétique en une chaîne de caractères qui correspond à l'expression arithmétique complètement parenthésée.*
2. *Affichez les chaînes de caractères qui correspondent aux expressions arithmétiques de la question* 2*.*

1.1.2 Interprétation

Nous allons maintenant écrire un interpréteur pour les expressions arithmétiques.

▸ **Question 4.** *La première étape est de définir un type* `valuation` *qui permet de représenter les valeurs associées à chaque variable qui apparaît dans une expression. Par exemple, un simple liste de couples dont le premier élément est le nom de la variable et le second élément est la valeur de cette variable peut faire l'affaire. Le module* `List` *d'OCaml contient tout ce qu'il faut pour manipuler de telles listes.*

▸ **Question 5.** *Définissez maintenant une fonction* `ainterp` *qui prend en arguments une expression arithmétique et une valuation et qui renvoie la valeur de cette expression.*

▸ **Question 6.** *À l'aide de votre fonction* `ainterp` *et de la valuation qui à x associe* 5 *et y associe* 9*, évaluez les expressions de la question* 2*.*

1.1.3 Substitutions

Lorsque l'on écrit une preuve d'un triplet de Hoare, on est amené à substituer une variable d'une expression arithmétique par une expression arithmétique (par exemple quand on traite une affectation).

▸ **Question 7.** *Écrivez une fonction* `asubst` *qui prend trois arguments ; le nom d'une variable, une expression arithmétique qui va remplacer cette variable et une expression arithmétique. La fonction* `asubst` *renvoie une expression arithmétique dans laquelle toutes les occurrences de la variable sont substituées par l'expression arithmétique donnée en argument.*

▸ **Question 8.** *À l'aide des fonctions* `asubst` *et* `aexp_to_string` *affichez les chaînes de caractères des expressions de la question* 2 *pour lesquelles les variables* x *et* y *sont respectivement substituées par les expressions* 7 *et* $z + 2$.

1.2 Les expressions booléennes

1.2.1 Syntaxe abstraite

Dans cette partie la syntaxe abstraite des expressions booléennes est définie. Aux deux constantes « vrai » et « faux », on ajoute les connecteurs logiques « et », « ou », la négation, l'égalité de deux expressions arithmétiques et la relation inférieure ou égale pour deux expressions arithmétiques.

▸ **Question 1.** *Donnez un type* `bexp` *qui donne la syntaxe abstraite des expressions booléennes de notre langage.*

▸ **Question 2.** *À l'aide du type* `bexp` *écrivez les expressions suivantes :*

- *vrai*
- *vrai et faux, non vrai, vrai ou faux*
- $2 = 4, 3 + 5 = 2 * 4, 2 * x = y + 1$
- $5 \leq 7, (8 + 9 \leq 4 * 5)$ *et* $(3 + x \leq 4 * y)$

▸ **Question 3.**

- *De même que pour les expressions arithmétiques, la lecture des expressions booléennes écrites avec la syntaxe abstraite n'est pas très commode. Écrivez une fonction* `bexp_to_string` *qui transforme une expression booléenne en une chaîne de caractères qui correspond à l'expression booléenne complètement parenthésée.*
- *Affichez les chaînes de caractères qui correspondent aux expression booléennes de la question* 2.

1.2.2 Interprétation

Nous allons maintenant écrire un interpréteur pour les expressions booléennes. Cet interpréteur va évaluer les expressions booléennes dans les booléens du langage de programmation utilisé. Pour cela nous devons utiliser une valuation afin d'obtenir les valeurs des variables arithmétiques que l'on peut rencontrer.

▸ **Question 4.** *Définissez une fonction* `binterp` *qui prend en arguments une expression booléenne et une valuation et qui renvoie un booléen qui est la valeur de vérité de l'expression booléenne.*

▸ **Question 5.** *À l'aide de votre fonction* `binterp` *et de la valuation qui à x associe* 7 *et à y associe* 3*, évaluez les expressions de la question* 2*.*

1.3 Les commandes du langage

1.3.1 Syntaxe abstraite

Dans cette partie on définit la syntaxe abstraite des expressions de programmation (ou programmes) de notre langage de programmation. Les différentes commandes qui permettent d'écrire les programmes sont la commande qui ne fait rien (`skip`), l'affectation de la valeur d'une expression arithmétique à une variable, la séquence de deux commandes, la conditionnelle et la boucle `repeat` qui exécute un certain nombre de fois un programme. Ce nombre de fois est donné par la valeur d'une expression arithmétique.

Par exemple, dans les boucles

```
repeat 10 do x := x+1 od
```

et

```
repeat 3*y do x := 2*x; z := z+1 od
```

les expressions arithmétiques 10 et $3 * y$ donnent le nombre d'exécutions des programmes `x := x+1` et `x := 2*x; z := z+1`. La boucle `repeat` est celle qui a été vue en cours.

▸ **Question 1.** *Donnez un type* `prog` *qui donne la syntaxe abstraite des programmes de notre langage.*

▸ **Question 2.** *À l'aide du type* `prog` *écrivez les programmes suivants :*

- `y := 7`
- `z := 3 + 4 ; x := 2* x`
- `n := 3; if (n <= 4) then n:= 2*n+3 else n:= n+1`
- `repeat 10 do x := x+1 od`

▸ **Question 3.**

- *Comme pour les expressions arithmétiques et booléennes, les programmes (i.e. les expressions de programmation) écrits à l'aide de la syntaxe abstraite sont difficiles à lire. Écrivez une fonction* `prog_to_string` *qui transforme un programme en une chaîne de caractères (cette chaîne de caractères peut contenir des sauts de ligne pour aider à la lecture).*
- *Affichez les chaînes de caractères qui correspondent aux programmes de la question* 2.

1.3.2 Interprétation

Nous allons maintenant écrire un interpréteur pour les programmes. Cet interpréteur va renvoyer une valuation qui contiendra toutes les valeurs des variables une fois le programme exécuté.

Pour cela nous utiliserons, bien sûr, une valuation, mais il faut aussi écrire une fonction `selfcompose` qui va réaliser l'exécution d'une boucle `repeat`.

Comme cela a été vu en cours, cette fonction `selfcompose` correspond à l'opération mathématique de composition d'une fonction par elle-même un certain nombre de fois. Pour bien comprendre, vous pouvez résoudre à la main la question 5 ci-dessous.

▸ **Question 4.** *Écrivez une fonction* `selfcompose` *qui prend en arguments une fonction des entiers vers les entiers (ou d'un type* `'a` *vers un type* `'a`*) et un entier n positif et qui renvoie une fonction des entiers vers les entiers (ou d'un type* `'a` *vers un type* `'a`*) qui est la composition de n fois par elle-même de la fonction donnée en argument.*
Remarques :

- *Parmi les langage autorisés pour le projet, OCaml est celui qui permet de manipuler le plus facilement les arguments qui sont des fonctions. Toutefois les parties programmation fonctionnelles (i.e. lambda expressions, fonctions anonymes) de Java et C++ vous permettent d'en faire autant.*
- *N'oubliez pas que quand $n = 0$, le résultat est la fonction identité (i.e celle qui pour un argument x renvoie x).*
- *Si vous êtes mal à l'aise avec le fait de renvoyer une fonction, vous pouvez ajouter un argument qui est la valeur sur laquelle la composition de fonction s'applique.*

▸ **Question 5.** *À l'aide de la fonction* `selfcompose`*, appliquez* 10 *fois la composition à elle-même de la fonction $f : x \to x + 2$ à la valeur* 0*. C'est-à-dire que l'on veut calculer $f^{10}(0) = \underbrace{f \circ \cdots \circ f}_{10 \text{ fois}}(0)$.*

▸ **Question 6.** *À l'aide de la fonction* `selfcompose` *et des fonctions* `ainterp` *et* `binterp`*, écrivez une fonction* `exec` *qui prend en arguments un programme et une valuation qui contient les valeurs initiales de toutes les variables du programme et qui renvoie une valuation qui contient toutes les valeurs finales des variables du programme.*

Remarque : La valuation renvoyée contient *les valeurs finales des variables. C'est-à-dire qu'elle peut en contenir plus, par exemple les valeurs des variables qui ont été calculées au cours de l'exécution du programme. Dans ce cas, pour une valuation, la valeur associée à une variable est celle de la première occurence rencontrée ou de la dernière occurence. Tout dépend de la façon dont l'interprétation de la commande d'affection met à jour les valeurs associées aux variables.*

▸ **Question 7.**

- *Écrivez un programme qui calcule la factorielle d'un entier positif et exécutez-le pour calculer la factorielle de* 5
- *Écrivez un programme qui calcule le nième nombre de la suite de Fibonacci (la version itérative) et exécutez-le pour calculer le* 8*ième nombre de cette suite.*

1.4 Triplets de Hoare et validité

Dans cette partie on définit la structure de données pour représenter un triplet de Hoare et on écrit une fonction qui permet de vérifier la validité d'un triplet pour une valuation.

Comme cela a été vu en cours, un triplet de Hoare est composé d'une précondition, d'un programme et d'une postcondition. Les pré et post conditions sont des formules logiques qui expriment des propriétés sur l'état de la mémoire. Généralement, il s'agit de formules logiques de la logique des prédicats (c'est la logique des raisonnements mathématiques) qui contient, entres autres, les quantificateurs « pour tout » et « il existe ».

Dans notre cas, comme nous l'avons fait en TP pour les formules logiques des triplets de Hoare, nous allons nous restreindre à la logique des propositions et, plus précisément, à la logique des propositions dont les variables propositionnelles sont remplacées par égalités et les inégalités sur les expressions arithmétiques de notre langage.

1.4.1 Syntaxe abstraite des formules de la logiques des propositions

Dans cette partie on définit la syntaxe abstraite des formules logiques. Aux deux constantes « vrai » et « faux », on ajoute les connecteurs logiques « et », « ou », « implique », la négation, l'égalité de deux expressions arithmétiques et l'inégalité inférieure ou égale pour deux expressions arithmétiques.

En procédant de cette façon, la syntaxe abstraite des formules logiques est un « décalque » de la syntaxe abstraite des expressions booléennes auquel on ajoute simplement le connecteur logique « implique ».

Remarquez aussi que l'on n'a pas de variables propositionnelles. Comme dit précédemment, ce sont les formules exprimant une égalité ou une inégalité entre deux expressions arithmétiques qui remplacent variables propositionnelles.

▸ **Question 1.** *Donnez un type* `tprop` *qui donne la syntaxe abstraite des formules logiques.*

▸ **Question 2.** *À l'aide du type* `tprop` *écrivez les formules suivantes :*

- *vrai*
- *vrai et faux, non vrai, vrai ou faux, faux implique vrai ou faux*
- $2 = 4, 3 + 5 = 2 * 4, 2 * x = y + 1$
- $3 + x \leq 4 * y, (5 \leq 7)$ *et* $(8 + 9 \leq 4 * 5)$
- $(x = 1)$ *implique* $(y \leq 0)$,

▸ **Question 3.**

- *Comme précédemment, les formules logiques écrites à l'aide de la syntaxe abstraite sont difficiles à lire. Écrivez une fonction* `prop_to_string` *qui transforme une formule logique en une chaîne de caractères qui correspond à la formule logique complètement parenthésée.*
- *Affichez les chaînes de caractères qui correspondent aux formules logiques de la question* 2.

1.4.2 Interprétation

Nous allons maintenant écrire un interpréteur pour les formules logiques. Cet interpréteur va évaluer les formules logiques dans les booléens du langage de programmation utilisé. Pour cela nous devons utiliser une valuation afin d'obtenir les valeurs des variables arithmétiques que l'on peut rencontrer.

▸ **Question 4.** *Définissez une fonction* `pinterp` *qui prend en arguments une formule logique et une valuation et qui renvoie un booléen qui est la valeur de vérité de la formule logique. Remarque : Pour interpréter l'implication* $P => Q$*, vous pouvez utiliser l'équivalence* $(P => Q) \equiv (\neg P) \vee Q)$.

▸ **Question 5.** *À l'aide de votre fonction* `pinterp` *et de la valuation qui à* x *associe* 7 *et à* y *associe* 3*, évaluez les expressions de la question* 2.

1.4.3 Substitutions

Lors de la manipulation des préconditions et des postconditions quand on écrit une preuve d'un triplet de Hoare, il est nécessaire de faire des substitutions dans les expressions arithmétiques contenues dans les préconditions et les postconditions. Il faut donc « remonter » les substitutions des formules logiques aux expressions arithmétiques.

▸ **Question 6.** *Écrivez une fonction* `psubst` *qui prend trois arguments ; le nom d'une variable, une expression arithmétique qui va remplacer cette variable et une formule logique. La fonction* `psubst` *renvoie une formule logique dans laquelle toutes les occurrences de la variable des expressions arithmétiques contenues dans la formule logique donnée en argument sont substituées par l'expression arithmétique donnée en argument. Par exemple, si dans la formule*

logique $x + 1 = 1$ *on remplace la variable* x *par l'expression arithmétique* $y * 3$*, on obtient l'expression logique* $3 * y + 1 = 1$*.*

▸ **Question 7.** *À l'aide des fonctions* `psubst` *et* `prop_to_string` *affichez les chaînes de caractères des formules logiques de la question 2 pour lesquelles les variables* x *et* y *sont respectivement substituées par les expressions* $3 * y$ *et* $k + 2$*.*

1.4.4 Les triplets de Hoare

Un triplet de Hoare est la donnée d'une précondition qui est une formule logique, d'un programme et d'une postcondition qui est une formule logique.

▸ **Question 8.** *Donnez un type* `hoare_triple` *dont les valeurs représentent un triplet de Hoare.*

▸ **Question 9.** *À l'aide du type* `hoare_triple` *écrivez les triplets de Hoare suivants :*

- $\{x = 2\}$ `skip` $\{x = 2\}$
- $\{x = 2\}$ `x := 3` $\{x \leq 3\}$
- $\{True\}$ `if x <= 0 then r := 0-x else r := x` $\{0 <= r\}$
- $\{in = 5$ *et* $out = 1\}$ `fact` $\{in = 0$ *et* $out = 120\}$ *où* `fact` *est votre programme de calcul de la factorielle de la variable* `in` *qui range le résultat dans la variable* `out`*.*

1.4.5 Validité d'un triplet de Hoare

Un triplet de Hoare est valide si pour toute valuation de ses variables, la précondition est satisfaite, le programme termine et la postcondition satisfait la valuation calculée par le programme.

Comme les pré et post conditions sont des formules de la logique des propositions, pour savoir si elles sont satisfaites pour une valuation, il suffit de les interpréter pour cette valuation et d'observer la valeur de vérité obtenue.

Par ailleurs, l'interpréteur d'un programme prend en entrée une valuation et renvoie la valuation calculée par le programme. De plus, on sait que, par construction, tous les programmes écrits dans notre langage terminent.

On a ainsi le moyen de vérifier la validité d'un triplet de Hoare pour une valuation. Il suffit de calculer l'interprétation de la précondition pour cette valuation, de calculer l'interprétation de la postcondition pour la valuation calculée par l'interprétation du programme et de former conjonction de ces deux interprétations pour obtenir la validité du triplet de Hoare.

On teste ainsi la validité d'un triplet de Hoare sur une valuation particulière. Bien évidemment pour vérifier la validité d'un triplet de Hoare, il faudrait la vérifier pour toutes les valuations, ce qui n'est pas réalisable dans un langage de programmation classique.

▸ **Question 10.** *Écrivez une fonction* `htvalid_test` *qui prend en argument un triplet de Hoare et une valuation et renvoie un booléen qui indique si le triplet est valide pour la valuation donnée en argument.*

2 Un (mini) prouveur en logique de Hoare

Le prouveur se décompose en deux parties. Une partie qui va s'occuper des preuves pour les formules de la logique des propositions (telles que spécialisées dans la partie précédente) et une partie qui va s'occuper des preuves pour les triplets de Hoare.

2.1 Les buts de preuves et le langage des tactiques

La première étape est de définir des structures de données pour représenter les buts de preuves et la syntaxe abstraite de notre langage de tactiques.

2.1.1 Les buts de preuves

Dans le cas particulier où nous nous sommes placés, un but de preuve est composé d'un contexte qui est une liste de formules de la logique des propositions étiquetées par un identificateur unique (qui est une chaîne de caractères) et d'une conclusion qui est soit un triplet de Hoare, soit une formule de la logique des propositions.

▸ **Question 1.** *Donnez le type* `goal` *qui permet de représenter un but de preuve.*

▸ **Question 2.** *À l'aide du type* `goal`*, écrivez les buts de preuve suivants :*

- *context :* $[H : (P \vee Q \Rightarrow R); H_2 : P]$ *conclusion :* $P \vee Q$
- *context :* $[]$ *conclusion :* $\{x = -3\}$ `if x <= 0 then 0-x else x` $\{x = 3\}$

▸ **Question 3.** *Écrivez une fonction* `print_goal` *qui permet d'afficher un but de façon similaire à l'exemple ci-dessous.*

$$
\begin{array}{l}
H : (P \vee Q \Rightarrow R) \\
H_2 : P \\
\texttt{==========} \\
P \vee Q
\end{array}
$$

Dans la suite, afin de pouvoir étiqueter de façon unique les formules du contexte, il faut écrire une fonction `fresh_ident` qui fournit une nouvelle chaîne de caractères à chaque appel. En OCaml, cette fonction peut s'écrire :

```
let fresh_ident =
  let prefix = "H" and count = ref 0
  in
```

```
function () ->  (count := !count + 1 ;
  prefix ^ (string_of_int (!count)))
```

Cette fonction prend comme argument l'unique valeur du type `unit` à savoir `()`. Vous avez sans doute déjà écrit des fonctions similaires en Java ou C++ (en TP de COO par exemple).

2.1.2 La règle de déduction pour la boucle `repeat`

Avant d'aborder la façon dont on peut implanter des tactiques qui vont correspondre à l'utilisation des règles de déduction de la logique des propositions et de la logique de Hoare, on doit établir une règle de déduction pour la boucle `repeat`.

Pour cela on va se ramener à une boucle `while` et utiliser la règle de déduction vue en cours pour cette boucle afin d'en déduire une règle pour la boucle `repeat`.

On commence par transformer la boucle `repeat` en boucle `while` avec une variable qui compte le nombre de tours de boucle. Ainsi le code

```
repeat e do c od
```

devient

```
i := 1;
while i <= e do
  c;
  i := i+1
od
```

On décore maintenant ce programme en utilisant la règle de la boucle `while` (pour rappel le notation `[e/x]P` signifie que l'on remplace toutes les occurrences de la variable x par l'expression e dans la formule logique P)

```
{[1/i]I]}
i := 1;
{I}
while i <= e do
  {I /\ (i <= e)}
  c;
  {[i+1/i]I}
  i := i+1
  {I}
od
{I /\ not(i <= e)}
```

Comme on sort de la boucle dès que $i > e$, on peut renforcer la postcondition en `{I /\ (i = e + 1)}` cela rend les preuves plus faciles à établir.

Ainsi la règle que l'on obtient est

$$\frac{\{I \wedge (i <= e)\}\ c\ \{[i+1/i]I\}}{\{[1/i]I\}\ \mathsf{repeat}\ e\ \mathsf{do}\ c\ \{I \wedge (i = e+1)\}}\ repeat(i)$$

où i est une variable de boucle non modifiée par `c`. On dit que i est une variable fantôme (ghost variable).

La règle est donc paramétrée par la variable fantôme. Dans le cas de boucles `repeat` imbriquées, il faut faire attention à ne pas utiliser la même variable fantôme car sinon la boucle interne modifie la variable fantôme et, de fait, ce n'est plus une variable fantôme (et on peut faire n'importe quoi).

Pour être utile à la preuve du triplet de Hoare, l'invariant I doit contenir au moins une occurrence de la variable fantôme, cela indique qu'il donne des informations sur l'exécution de la boucle.

À titre d'exemple, prouvons le triplet $\{x = y\}$ `repeat 10 do x:= x+1 od` $\{x = y + 10\}$. Il faut commencer par chercher une formule logique qui reste vraie, avant la boucle, à chaque tour de boucle et après la boucle. En prenant i comme variable fantôme, considérons les valeurs des variables i, x et de l'expression $y + i - 1$ à chaque tour de boucle, on obtient le tableau :

tour	i	x	$y + i - 1$
0	1	y	y
1	2	$y + 1$	$y + 1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
9	10	$y + 9$	$y + 9$
10	11	$y + 10$	$y + 10$

On voit que l'égalité $x = y + i - 1$ reste vraie avant la boucle, à chaque tour de boucle et après la boucle. Il s'agit donc d'un bon candidat pour l'invariant I.

Il faut maintenant décorer le programme selon la règle de la boucle `repeat` en utilisant notre candidat invariant

```
{x = y}
{x = y + 1 - 1}
repeat 10 do
  {(x = y + i - 1) /\ (i <= 10)}
  {(x + 1) = y + (i + 1) - 1}
    x := x + 1
  {x = y + (i + 1) + 1}
od
{(x = y + i - 1) /\ (i = 10 + 1)}
{x = y + 10}
```

Il ne reste plus qu'à appliquer les règles de déduction pour prouver le triplet.

▸ **Question 4.** *Donnez l'arbre de preuve qui montre la suite des règles de déduction à appliquer pour prouver le triplet précédent.*

▸ **Question 5.** *En donnant l'arbre de preuve, prouvez le triplet*
$\{(r = 0) \wedge (n = 1)\}$ `repeat 5 do r := r + n; n := n + 1 od` $\{(r = 15) \wedge (n = 6)\}$
pour trouver l'invariant, observez les variables r et n, la variable fantôme i et l'expression

arithmétique $i * (i - 1)$. L'invariant se compose d'une conjonction (i. e. un « et ») de deux égalités, l'une entre r et i, l'autre entre n et i. Décorez ensuite le programme et écrivez l'arbre de preuve.

2.1.3 Le langage des tactiques

Le langage des tactiques est défini par sa syntaxe abstraite. Pour la partie « logique des propositions » on retrouve les tactiques de la déduction naturelle utilisées dans le TP `DedNat.v` à savoir :

```
And_Intro, And_Elim_1, And_Elim_2, Or_Intro_1, Or_Intro_2, Or_Elim,
Impl_Intro, Impl_Elim, Not_Intro, Not_Elim.
```

À cette liste nous ajoutons la tactique `Assume`, aussi utilisée dans `DedNat.v`, qui permet d'introduire une nouvelle hypothèse dans un contexte afin d'obtenir une preuve d'une formule. Par la suite, il faut aussi donner une preuve de cette hypothèse.

On ajoute également la tactique `Exact` qui permet d'utiliser une hypothèse du contexte pour prouver une conclusion.

Enfin, on ajoute la tactique `Admit` qui permet d'admettre une égalité ou une inégalité entre deux expressions arithmétiques. Cette dernière tactique est une simplification importante car elle permet de se passer de la fameuse tactique `lia` qui implante la procédure de décision pour l'arithmétique linéaire et elle évite aussi de faire toutes les preuves liées aux propriétés des entiers. En ce sens, notre prouveur est grandement simplifié.

Pour la partie « logique de Hoare » on retrouve aussi les tactiques utilisées en TP. Pour faciliter l'écriture de la syntaxe abstraite voici les noms de ces tactiques :

```
HSkip, HAssign, HCons, HSEq, HIf
```

À cette liste on ajoute la tactique `HRepeat` qui correspond à la règle de déduction pour la boucle `repeat` de notre langage.

▸ **Question 6.** *Donnez un type `tactic` qui définit la syntaxe abstraite des tactiques. Pour les tactiques de la partie « logique des propositions », vous devez tenir compte que les tactiques*

```
And_Intro, Or_Intro_1, Or_Intro_2, Impl_Intro, Not_Intro.
```

qui s'appliquent à la conclusion du but sur lequel elles agissent, ne prennent pas d'argument et que les tactiques

```
And_Elim_1, And_Elim_2, Or_Elim, Impl_Elim, Not_Elim, Exact.
```

qui s'appliquent à des formules du contexte, prennent, selon le cas, un ou deux arguments qui sont les identificateurs des hypothèses du contexte. Enfin la tactique `Assume` prend une formule logique en argument.
Pour la partie « logique de Hoare », vous devez tenir compte que les tactiques

```
HSkip, HAssign, HIf
```

ne prennent pas d'argument, que la tactique

```
HRepeat
```

prend une chaîne de caractères en argument (le nom de la variable fantôme) et que les tactiques

```
HCons , HSEq
```

prennent respectivement deux arguments et un argument qui sont des formules logiques qui servent à créer de nouveaux buts. Remarquez que, contrairement aux TP, il n'y pas l'équivalent des tactiques `Hoare_consequence_rule_left` *et* `Hoare_consequence_rule_right`*. La tactique* `HCons` *implante la règle de la conséquence de la logique de Hoare en agissant à la fois sur la précondition et la postcondition.*

2.2 Appliquer une tactique à un but

L'objectif de cette partie est d'écrire une fonction `apply_tactic` qui applique une tactique à un but pour réaliser une étape d'une preuve. Le principe général est de considérer le cas de chaque tactique et de construire la liste des nouveaux buts de preuve en fonction de l'action de la tactique. Il y a plusieurs manières de faire cela. Une approche simple est de filtrer selon la syntaxe abstraite des tactiques et, si nécessaire en fonction de la tactique, selon la conclusion du but à traiter.

Selon la tactique utilisée, celle-ci génère un ou plusieurs buts, en créant un nouveau contexte et/ou une nouvelle conclusion à prouver.

Par exemple, pour la partie « logique des propositions », la tactique `And_Intro` s'assure que la conclusion du but est bien une formule logique qui est une conjonction (i. e. un « et ») et génère deux buts ayant le contexte d'origine et avec respectivement chaque formule de la conjonction à prouver.

La tactique `And_Elim_1` recherche dans le contexte l'hypothèse correspondant à l'identificateur donné en argument de la tactique, s'assure que la formule de l'hypothèse est bien une conjonction en la filtrant et crée un nouveau but avec la première formule de la conjonction ajoutée dans le contexte d'origine et pour la conclusion la même que celle du but d'origine.

La tactique `Impl_Intro` s'assure que la conclusion du but est bien une formule logique qui est une implication et crée un nouveau but avec la première formule de l'implication ajoutée au contexte d'origine et avec la seconde formule de l'implication comme conclusion.

La tactique `Admit` s'assure que la conclusion du but est, soit une égalité, soit une inégalité de deux expressions arithmétiques et, dans ces deux cas, renvoie une liste vide de buts. On termine ainsi la preuve en admettant l'égalité ou l'inégalité présente dans la conclusion du but. C'est brutal, mais cela permet de ne pas aller trop loin dans le développement du prouveur.

Le principe est le même pour les autres tactiques. Pour la partie « logique de Hoare » ce principe reste identique. Par exemple, la tactique `HSkip` s'assure que la conclusion est bien un triplet de Hoare dont la commande est `skip` et vérifie que la postcondition est égale à la précondition. Si cela est bien le cas, la liste des buts renvoyée est la liste vide (i.e. on a fini la preuve) sinon la tactique échoue.

Pour vérifier l'égalité de la postcondition et la précondition, on utilise l'égalité structurelle

(i.e. on vérifie que les deux valeurs ont la même structure). Selon les langages, elle est soit disponible de base (c'est le `=` de OCaml), soit il faut la programmer (c'est un opérateur `==` d'une classe C++).

Pour la tactique `HIf`, il y a une petite difficulté. Il faut convertir l'expression booléenne qui apparaît dans la conditionnelle en une formule logique dans les préconditions des deux nouveaux buts à prouver.

▸ **Question 1.** *Écrivez une fonction* `bool2prop` *qui convertit une expression booléenne en formule logique.*

Pour la tactique `HRepeat`, il faut s'assurer que la postcondition est de la bonne forme (i.e $I \wedge (i = e + 1)$) et que la precondition soit bien égale à $[1/i]I$ où i est la variable fantôme (i.e. celle dont le nom est donné en argument de la tactique `HRepeat`).

▸ **Question 2.** *Il s'agit maintenant d'écrire la fonction* `apply_tactic`*. Vous pouvez soit écrire une fonction qui analyse tous les cas de filtrage soit écrire deux fonctions* `apply_hoare_tactic` *et* `apply_prop_tactic` *qui traitent respectivement la partie « logique de Hoare » et la partie « logique des propositions » qui s'appellent mutuellement pour traiter les tactiques qui ne sont pas prises en compte dans l'une ou l'autre. Vous écrivez ensuite la fonction* `apply_tactic` *qui appelle soit* `apply_hoare_tactic` *soit* `apply_prop_tactic` *selon la conclusion du but à traiter.*

2.2.1 La logique des propositions

Il est désormais possible de programmer pas à pas une preuve d'une formule logique. Remarquez qu'il est bien demandé de programmer la preuve, il ne s'agit pas de faire une boucle d'interaction et/ou de mettre en place une analyse lexicale.

▸ **Question 3.** *Utilisez votre fonction* `apply_tactic` *pour prouver le but suivant en programmant l'application successive des tactiques :*

$$\frac{}{(P \vee Q \Rightarrow R) \Rightarrow (P \Rightarrow R) \wedge (Q \Rightarrow R)}$$

2.2.2 La logique de Hoare

▸ **Question 4.** *Utilisez votre fonction* `apply_tactic` *pour prouver les triplets de Hoare suivants en programmant l'application successive des tactiques :*

- $\{x = 2\}$ `skip` $\{x = 2\}$
- $\{y + 1 < 4\}$ `y := y + 1` $\{y < 4\}$
- $\{y = 5\}$ `x := y + 1` $\{x = 6\}$
- $\{True\}$ `z := x; z := z+y; u := z` $\{u = x + y\}$
- $\{True\}$ `if v <= 0 then r := 0-v else r := v` $\{0 <= r\}$
- $\{x = y\}$ `repeat 10 do x:= x+1 od` $\{x = y + 10\}$

▸ **Question 5.** *Pour les triplets que vous avez prouvé, utilisez la fonction* `htvalid_test` *pour tester leur validité sur plusieurs exemples de valuation.*

Si vous trouvez une valuation pour laquelle le triplet n'est pas valide, c'est que votre preuve est fausse. Par exemple, une égalité ou une inégalité entre deux expressions arithmétiques qui a été admise était, en fait, fausse ou bien (et c'est plus ennuyeux) une tactique implante mal une règle.

Voilà, c'est fini ! Vous imaginez bien ce qu'il reste à faire. Par exemple, mettre en place l'application d'une succession d'applications de tactiques ou mettre en place une syntaxe concrète avec une analyse syntaxique et lexicale ou encore mettre en place une boucle d'interaction, etc.