Etienne Binginot
TP 1

Compte-rendu Projet LSI

Les arbres d’intervalles

Sommaire

Notice d'utilisation

La bibliothèque est composée de 3fichiers. Le fichier intervalle.ml permet de gérer les intervalles d'éléments quelconques, le fichier arbreRN.ml les arbres d'éléments quelconques et le fichier arbre_intervalles.ml les arbres d'intervalles d'éléments quelconques.

Fichier intervalle.ml

Le fichier intervalle.ml contient deux interfaces : une pour les intervalles, et une autre pour les éléments de ces intervalles (un type ordonné).
Dans ce fichier on retrouve également un foncteur MakeIntervalle qui utilise un typeOrdonne pour fonctionner, créant ainsi un module intervalle sur ce type.

Le module créé avec le foncteur MakeIntervalle possède les fonctions (principales) suivantes :

- La fonction inter permet de créer un intervalle valide en lui donnant deux bornes et deux booléens indiquant la fermeture des bornes (true → fermé, false → ouvert).

- La fonction est_dans_intervalle permet d'indiquer la présence d'un élément donné dans un intervalle sur cet élément donné.

- La fonction comp permet de comparer des intervalles, en renvoyant une valeur négative si le premier intervalle est plus petit, positive si plus grand, égale si même taille.

- La fonction sont_disjoints indique si les deux intervalles donnés en paramètre sont disjoints.

- La fonction union réalise l'union de deux intervalles. Le résultat peut être un seul intervalle ou une union d'intervalle (résultat sous forme Either.t, pouvant donc être un couple ou un unique intervalle)

- La fonction valide_inter permet de créer plusieurs intervalles aléatoires, et de réaliser des opérations sur ceux-ci pour tester leur validité. Testé et approuvé !

Fichier arbreRN.ml

Le foncteur MakeArbre d'arbreRN.ml prend un lui aussi un TypeOrdonne, et va réaliser un module arbre sur cet élément.

Le module créé avec le foncteur MakeArbre possède les fonctions (principales) suivantes :

- La fonction vide permet de créer un Arbre vide, qui pourra ensuite être complété.
- La fonction insérer permet d'ajouter un élément dans un arbre.
- La fonction supprimer permet de retirer un élément dans un arbre.
- La fonction est_bicolore permet d'indiquer si un arbre est bicolore ou non.

Fichier arbre_intervalles.ml

Le foncteur MakeArbreIntervalle d'arbre_intervalles.ml prend lui un module intervalle provenant de MakeIntervalle en paramètres, et réalise avec celui-ci un arbre sur les intervalles, ceux-ci ayant comme éléments les mêmes que les éléments du module intervalle donné en paramètre.

Le module créé avec le foncteur MakeArbreIntervalle possède les fonctions (principales) suivantes :

- Il possède déjà toutes les fonctions de l'interface Arbre, et donc du foncteur MakeArbre.
- La fonction est_dans_arbre_intervalle permet d'indiquer si un élément se trouve dans un des intervalles de l'arbre.
- La fonction ajout permet d'ajouter de façon valide un intervalle dans un arbre d'intervalle. Attention, la fonction insérer existe encore mais celle-ci donnera un arbre d'intervalle invalide !
- La fonction retrait permet de retirer de façon valide un intervalle d'un arbre d'intervalle. Attention, la fonction supprimer existe encore mais celle-ci donnera un arbre d'intervalle invalide !
- La fonction valide_arbre_inter permet quant à elle de créer un arbre d'intervalle aléatoire et de réaliser des opérations sur celui-ci pour tester sa validité. Testé et approuvé !

Choix d'implantation

J'ai choisi d'utiliser les foncteurs pour implémenter mes modules, ce qui me permet d'avoir des types polymorphes. Afin de réaliser mon foncteur et mon interface sur les arbres d'intervalle, j'ai choisi de directement récupérer le code de mon interface d'arbre pour mon interface d'arbre d'intervalles, et de créer un module utilisant MakeArbre et prenant en élément les intervalles afin d'avoir une base sur laquelle travailler, et ainsi éviter d'avoir à dupliquer du code. En effet, avec un include, il est directement possible d'importer le code sans avoir à le réécrire.

Fichier intervalle.ml

J'ai choisi de créer des opérateurs directement basés sur la fonction de comparaison des éléments de mes intervalles pour très largement simplifier mon code : ainsi, je n'ai pas besoin de comparer le résultat de ma comparaison avec 0 pour connaitre le résultat.

Dans la fonction est_dans_intervalle, j'ai choisi de seulement traiter tous les cas où mon élément est dans l'intervalle ainsi. Ainsi, en traitant ces cas, je sais que dans tous les autres cas l'élément ne se trouve pas dans l'intervalle et je peux renvoyer false.

Dans la fonction de comparaison des intervalles, j'ai regroupé ensemble les cas où i1 est plus grand que i2 et inversement. Ainsi, je traite tous les cas où je renvoie 1 ou -1 et je sais que je peux renvoyer 0 dans tous les cas autres cas (hors cas Vide qui renvoie une erreur). De plus, j'ai choisi de réaliser mes cas comme ça et de ne pas seulement traiter le cas où je renvoie 1 puis 0 et -1 dans tous les autres cas car bien que cela réduise le nombre de cas, en traitant 1 et -1, je sais qu'ensuite j'ai l'égalité entre les éléments que j'ai comparé si un résultat n'a pas été renvoyé. Par exemple, je compare x1 $>$ x2 et x2 $>$x1, si aucun des deux n'a renvoyé un résultat je sais que x1 $=$ x2.

Dans la fonction sont_disjoints, j'ai choisi d'appliquer la même stratégie que dans est_dans_intervalle. Je vérifie tous les cas où deux intervalles sont disjoints, et si je ne suis rentré dans aucun des cas, c'est qu'ils ne sont pas disjoints donc je renvoie false pour tout le reste.

Dans la fonction union, j'ai choisi d'utiliser des méthodes outils comme bornemin, bornemax pour largement simplifier mon code. De plus, j'utilise également beaucoup la fonction sont_disjoints. En effet, il est très simple de réaliser l'union de deux intervalles disjoints, il suffit de renvoyer le couple contenant les deux intervalles.

La fonction random_intervalle permet d'elle-même de réinitialiser sa graine d'aléatoire pour être sûr d'avoir un aléatoire différent à chaque fois.

J'ai dû réaliser de nombreuses fonctions d'inclusions différentes en fonction des cas. En effet, bien que l'inclusion entre deux ensembles i1 et i2 soit assez évidente (différence

== Vide), il faut aussi être capable de calculer l'inclusion entre un ensemble i1 et un ensemble i4 sous forme d'union (is_included_union).

Les fonctions is_included_either et is_included_either_2 sont là pour ça, elles vont appeler, en fonction que le premier ou le second élément soit du type Either, la bonne fonction d'inclusion (il faut traiter le cas où on se retrouve avec un couple !).

La fonction valide_diff utilise elle un fonctionnement un peu particulier. Pour savoir si les valeurs de i1 sont exclusivement soit dans i2, soit dans i4, on regarde déjà si les valeurs de i4 sont dans i1. Ainsi, on sait logiquement qu'une partie des valeurs de i1 se trouve dans i4 (potentiellement la totalité). Ensuite, il faut regarder si i4 et i2 sont bien disjoints. Si c'est le cas, si ce qui reste des valeurs de i1 (i1 – i4) est dans i2, cela veut dire que la différence est validée !

Les fonctions qui suivent valide_inter sont-elles des fonctions outils servant dans arbre_intervalles.ml. En effet, le type Inter étant défini dans intervalle.ml, c'est le seul endroit où on peut utiliser sa syntaxe.

De plus, les fonctions bornemin et bornemax sont redéfinies ici. Cela permet de coller à la signature de la fonction dans l'interface, et vu que ceux sont des fonctions outils ce n'est pas bien grave, les anciennes bornemin et bornemax ont déjà rempli leur rôle au-dessus. J'ai séparé les anciennes bornemin et bornemax en deux fonctions chacune (la nouvelle bornemin et borneminFerme, de même pour le max) car cela permettait de légèrement simplifier le code dans arbre_intervalles.ml, au prix de quelques signatures en plus ici.

Fichier arbreRN.ml

Dans ce fichier, j'ai placé ce que l'on avait réalisé en tp dans un foncteur MakeArbre. J'y ai remis toutes les fonctions les plus importantes sur les arbres, pour avoir un type polymorphe fonctionnel sur les arbres bicolores. De plus, j'ai rajouté quelques signatures dans l'interface Arbre afin de pouvoir efficacement les utiliser où je veux.

Attention cependant, bien qu'il soit techniquement possible de lui donner un type intervalle pour fonctionner, cela ne donnera pas des arbres d'intervalles fonctionnels. En effet, l'algorithme d'ajout et de retrait étant différent, les fonctions insérer et supprimer ne sont pas utilisables avec les arbres d'intervalles, et cela casse quelques autres fonctions qui les utilisent comme union ou différence par exemple.

Fichier arbre_intervalles.ml

J'ai tout d'abord choisi de récupérer pas mal de fonctions de mon module intervalle en paramètre du foncteur sous forme de fonction classique, sans avoir à utiliser le NomDuModule.LaFonction. Cela permet de légèrement simplifier le code, étant donné que ceux sont des fonctions que j'utilise souvent.

Pour faire fonctionner est_dans_arbre_intervalle, j'ai choisi d'utiliser la fonction est_dans_intervalle de mon module intervalle. Ainsi, si je sais qu'il n'est pas dans

l’intervalle du nœud actuel de l’arbre, et qu’il n’est pas plus petit que la borne min de l’intervalle, alors il est forcément plus grand et donc on doit faire un appel récursif à droite.

Pour beaucoup de fonctions, j’ai choisi d’utiliser les fonctions fold_left et fold_right du module List. Cela permet d’éviter d’avoir à faire des versions récursives parfois lourdes pour pas grand-chose, en choisissant les bonnes fonctions à lui donner. Si nécessaire, j’ai aussi utilisé la fonction flip du module Fun pour rendre les fonctions utilisables avec les fonctions fold.

De plus, j’ai redéfini les fonctions union, intersection et différence pour les rendre compatible avec les arbres d’intervalles. Pour cela, il suffit de reprendre le même algorithme mais en changeant les insérer/supprimer en ajout/retrait.

J’ai également créé une fonction d’affichage d’arbre se basant sur les affichages d’inter, me permettant d’efficacement regarder l’état de mes arbres issus du module. En effet, sans ces fonctions, il est difficile de bien se représenter l’état des objets que l’on manipule, ceux-ci étant décrit par utop comme étant abstrait.

Problèmes rencontrés

Afin de pouvoir importer le code du foncteur d'arbreRN.ml et de son interface, il a fallu de nombreuses heures de test pour trouver un moyen de le réaliser. Il y a eu en effet de nombreux problèmes de compatibilités de code, les types ab de arbre_intervalle.ml et de arbreRN.ml étant considérés différents à la base. La solution a été de directement définir le type ab dans l'interface Arbre, puis d'importer via un include le code d'Arbre dans IntervalleArbre. Ainsi, le type est défini qu'une seule fois et il n'y a pas de problème de compatibilité.

Pour récupérer le code du foncteur d'arbreRN.ml, il a fallu définir un module utilisant ce foncteur avec comme paramètre un module typeOrdonné sur les intervalles. En effet, je me suis très vite rendu compte qu'il était impossible d'importer le code d'un foncteur, il a donc fallu utiliser cette petite technique pour récupérer le code tout en le laissant compatible avec ce que j'avais déjà.

Il y a également eu de nombreux problèmes pour rendre les foncteurs compatibles, en effet, il a fallu choisir les bonnes fonctions à mettre dans la signature de IntervalleI pour pouvoir utiliser ces fonctions dans MakeIntervalleArbres. En effet, il a fallu utiliser de nombreuses fonctions annexes car le type inter n'étant pas défini dans MakeIntervalleArbres, il n'était pas possible de réaliser des match utilisant la syntaxe de type inter (Inter (x, y, bx, by) par exemple).

Preuves

Preuve de inter_non_disjoints :

Induction sur le cardinal des arbres d'une liste d'arbre. On note l'ordre <[A] et on note l'ensemble des listes d'arbre [A]. (ordre bien fondé car il s'agit de l'ordre <, on compare les cardinaux)

Propriété : La fonction se finit et renvoie le bon résultat (l'ensemble des intervalles non disjoints de la liste avec i)

Cas de base : Liste vide
Alors on renvoie directement l'accumulateur, qui de base vaut la liste vide. Notre liste de base étant vide, il n'y a pas d'intervalles ne pouvant pas être disjoints avec i donc on renvoie le bon résultat et l'appel se termine bien.

Induction : Soit w € [A] \ [] tel que pour tout v € [A] vérifiant v<[A]w, v vérifie la propriété. W étant non vide, il y a deux cas possibles :

- Le premier élément de w est l'arbre vide. Alors la fonction réalise un appel récursif en gardant les mêmes valeurs pour les paramètres sauf pour la liste où le premier élément est retiré. On appelle z cette liste. Alors z<[A]w et donc par induction, l'appel récursif se termine et renvoie le bon résultat. L'arbre vide ne pouvant pas ne pas être disjoints avec i, on renvoie bien le bon résultat et l'appel se termine.

- Le premier élément de w est un arbre non vide t. Deux cas :

    . Si l'intervalle associé à t est disjoint avec i alors on réalise un appel récursif en gardant les mêmes paramètres sauf pour la liste qui devient notre ancienne liste sans t + l'arbre gauche de t + l'arbre droit de t. On appelle cette liste z. Alors z <[A]w et par induction, l'appel se termine bien et renvoie le bon résultat. De plus, l'intervalle associé à t étant disjoints avec i, il faut bien ne pas l'ajouter au résultat, donc cela fonctionne.

    . Sinon, on réalise un appel récursif en ajoutant notre intervalle à l'accumulateur et en reprenant la même liste z qu'au-dessus. De même, z <[A]w et par induction, l'appel se termine bien et renvoie le bon résultat. De plus, l'intervalle associé à t étant non disjoints avec i, celui-ci est bien ajouté au résultat renvoyé donc cela fonctionne.

Par induction, la propriété est donc vraie pour tout w € [A] et donc l'appel à la fonction se termine et renvoie bien le bon résultat.

Preuve de retire_list :

La fonction prend en paramètre un arbre et une liste. Deux cas :
- Si la liste est vide, on renvoie l'arbre sans modification donc l'appel se termine bien et renvoie le bon résultat (un arbre moins rien donne bien le même arbre).
- Sinon, on supprime un à un les éléments de la liste de l'arbre. Or, on sait que les éléments de la liste sont forcément dans l'arbre, et que la fonction supprimer renvoie le bon résultat. Ainsi, on renvoie bien le bon résultat et l'appel se termine.

Preuve de ajout :

La fonction ajout prend un intervalle i à ajouter à un arbre a.

a) On sait que la fonction inter_non_disjoints renvoie bien le bon résultat, ainsi l correspond bien à la liste des intervalles non disjoints des intervalles de a avec i.

b) De plus, on sait que la fonction retirer_list renvoie bien le bon résultat donc a correspond bien à l'ancien arbre a sans les intervalles non disjoints avec i. Ainsi, l'intervalle i est disjoint avec tous les intervalles du nouvel arbre a.

c) On réalise la preuve de l'appel (List.fold_left union_nondisjoints i l).
union_nondisjoints réalise l'union de deux intervalles supposés non disjoints. S'ils sont disjoints, la fonction renvoie une erreur.
On sait que tous les intervalles de l sont non disjoints avec i. Ainsi, l'union de i et du premier élément de l va être réalisé sans problème, car on sait que l'union renvoie le bon résultat.
Ensuite, on va réaliser l'union du résultat obtenu et du deuxième intervalle de l s'il existe. On sait que l'utilisation de la fonction union_nondisjoints est possible car si i est non disjoints avec cet intervalle, l'union de i et de n'importe quoi sera encore non disjoints avec cet intervalle.
Ainsi, l'appel se termine bien et renvoie le bon résultat.

d) Finalement, on insère cet intervalle dans l'arbre.
Comme tous les intervalles non disjoints avec i ont été supprimé de l'ancien arbre, on sait que notre intervalle est disjoint avec tous les intervalles de l'arbre a.
De plus, on sait qu'insérer renvoie bien l'arbre avec l'élément inséré dedans (prouvé en tp).

Ainsi, on sait qu'ajout se termine bien et renvoie le bon résultat.

Preuve de retrait :

La fonction retrait prend un intervalle i et un arbre a.

a) On sait que la fonction inter_non_disjoints renvoie bien le bon résultat, ainsi l correspond bien à la liste des intervalles non disjoints des intervalles de a avec i.

b) De plus, on sait que la fonction retirer_list renvoie bien le bon résultat donc a correspond bien à l'ancien arbre a sans les intervalles non disjoints avec i. Ainsi, l'intervalle i est disjoint avec tous les intervalles du nouvel arbre a.

c) Ensuite, on réalise la différence entre chaque intervalle de l et i. On sait que la différence se termine bien et renvoie le bon résultat, donc on récupère bien la liste l' des intervalles de l moins i.

d) Cependant, cette liste doit être traitée.
En effet il ne faut pas ajouter le vide, et les Either.t doivent être transformés en type intervalle classique, et les couples séparés. Pour cela, on utilise la fonction liste_intervalle_retrait de notre module intervalle.

Induction sur la taille de l (on utilise l'ordre bien-fondé <) avec l € [I]

Propriété : la fonction se finit et renvoie le bon résultat.

Cas de base : l'appel de liste_intervalle_retrait sur la liste vide renvoie la liste vide. L'appel se termine bien et il s'agit du bon résultat (pas d'intervalle à traiter donc on ne renvoie rien).

Induction : Soit w € [I] / [] tel que pour tout v € [I] vérifiant card(v) < card(w), v vérifie la propriété.
Appel de la fonction sur w, alors comme w est non vide, 3cas :

- Le premier élément de w est vide. Dans ce cas, on réalise un appel récursif sur w sans son premier élément, sans ajouter le vide à l'accumulateur. On appelle w sans son premier élément z.
Or card(z) < card(w). Donc l'appel récursif se termine bien et renvoie le bon résultat, et celui-ci ne contient bien pas notre élément qui est vide.

- Le premier élément de w est un unique intervalle. Dans ce cas, on réalise un appel récursif sur w sans son premier élément, en ajoutant notre intervalle à l'accumulateur. On appelle w sans son premier élément z.
Or card(z) < card(w). Donc l'appel récursif se termine bien et renvoie le bon résultat, et celui-ci contient bien notre élément car il a été ajouté à l'accumulateur.

- Le premier élément de w est un couple d'intervalle. Dans ce cas, on réalise un appel récursif sur w sans son premier élément, en ajoutant nos deux intervalles à l'accumulateur. On appelle w sans son premier élément z.

Or card(z) < card(w). Donc l'appel récursif se termine bien et renvoie le bon résultat, et celui-ci contient bien nos 2 intervalles car ils ont été ajoutés à l'accumulateur.

Par induction, la propriété est donc vraie pour tout l € [I] et donc l'appel à la fonction se termine bien et renvoie le bon résultat.

e) Notre liste l' contient donc bien tous les éléments de l'ancien l' sans la liste vide, avec les couples d'intervalles séparés (plus de type Either.t).
Ainsi, chaque élément de l' est disjoint avec i, chaque intervalle de l'arbre est disjoint avec i. On ajoute un à un les éléments de l' à l'arbre. On sait que les éléments de l' sont disjoints avec les éléments de l'arbre car l'arbre de base est valide. Donc chaque élément de l'arbre est disjoint avec les autres éléments de l'arbre. Or, si un intervalle i est disjoint avec un intervalle j, la différence de i avec n'importe quel autre intervalle sera encore disjoint avec j. Ainsi, l'arbre obtenu est bien valide.

Donc l'appel à retrait se termine bien et renvoie le bon résultat.

Preuve de est_dans_intervalle :

Trivialement, il y a 3cas où un élément peut être inclus dans un intervalle :
- Il est supérieur strictement à la borne min et est inférieur strictement à la borne max
- Il est égal à la borne min et celle-ci est fermée
- Il est égal à la borne max et celle-ci est fermée
Dans tous les autres cas, notre élément n'est pas inclus dans l'intervalle.

Or, si on regarde le fonctionnement de notre fonction, on voit qu'on a différents cas. Soit i notre intervalle de la forme Inter (x, y, bx, by) et e notre élément.
- si e > x && e < y alors on renvoie true. On remarque que cela correspond bien au cas 1, donc celui-ci est traité
- si (e $=$ x && bx) || (e $=$ y && by) alors on renvoie true. On remarque que cela correspond aux deux autres cas, donc ceux-ci sont traités.
- sinon on renvoie false. On voit que cela correspond bien à notre algorithme qui nous expliquait que les 3cas au-dessus étaient les seuls pouvant correspondre à l'appartenance.
Ainsi, notre fonction traite bien tous les cas et renvoie donc le bon résultat.

Preuve de sont_disjoints :

Il y a plusieurs cas où deux intervalles peuvent être disjoints :
- Si l'un des deux vaut l'intervalle Vide, alors ils sont disjoints
- Si la borne min d'un des intervalles est strictement supérieure à la borne max de l'autre, alors ils sont disjoints
- Si la borne min d'un des intervalles est égale à la borne max de l'autre mais qu'au moins une des deux bornes est ouverte, alors ils sont disjoints
- Sinon ils ne sont pas disjoints

Si on regarde notre fonction, on remarque que celle-ci possède plusieurs cas sur i1, i2:
- Quand la borne min de i1 est plus grande que la borne max de i2 on renvoie true. Inversement, quand la borne min de i2 est plus grande que la borne max de i1 on renvoie true. Ces deux cas ensemble correspondent bien au deuxième cas de l'algorithme.
- Quand la borne min de i1 est égale à la borne max de i2 et que l'un des deux intervalles est ouvert au niveau de l'égalité ou que les deux sont ouverts au niveau de l'égalité, on renvoie true. Ensuite on réalise le cas symétrique en inversant les rôles de i1 et i2. Ces deux cas ensemble correspondent bien au troisième cas de l'algorithme.
- Si l'un des deux intervalles est vide, on renvoie true. Cela correspond bien au premier cas de l'algorithme
- Sinon on renvoie false, dernier cas de l'algorithme.

Ainsi, notre fonction traite bien tous les cas de l'algorithme et renvoie donc le bon résultat.

Preuve de comp :

Pour comparer des intervalles, nous utilisons l'algorithme du sujet : inter x y b_x b_y est plus petit que inter x' y' b_x' b_y' si :

- x est strictement plus petit que x',
- x est égal à x' et b_x' est strictement plus petit que b_x,
- (x, b_x) est égal à (x', b_x') et y est strictement plus petit que y',
- (x, b_x, y) est égal à (x', b_x', y') et b_y est plus petit que b_y'.

Prouvons que notre fonction réalise bien cet algorithme. Celui-ci comporte 10cas :

- Tout d'abord, si un des deux intervalles vaut l'intervalle Vide, on renvoie une erreur. En effet, il est impossible de comparer un intervalle avec le Vide ! (On suppose dans cet algorithme qu'on ne peut pas comparer le Vide avec le Vide).

- Les deux prochains cas correspondent à la comparaison des bornes min des intervalles. Si i1 possède une plus grande borne min, il est plus grand. Si l'inverse est vérifié, plus petit. Ainsi, nous réalisons bien le premier cas de l'algorithme, et on sait maintenant que si nous n'avons pas à ce stade renvoyé un résultat, les bornes min sont égales

- Ensuite, on compare les ouvertures des intervalles, si la borne min de i1 est fermée alors que celle de i2 est ouverte, i1 est plus petit que i2. Si l'inverse est vérifié, i2 est plus petit que i1. Ainsi, ces deux cas correspondent bien au deuxième cas de l'algorithme, et on sait maintenant que les ouvertures des bornes mins sont les mêmes.

- Le sixième et le septième cas correspondent à la comparaison des bornes max. Si la borne max de i1 est plus petite que celle de i2, alors i1 est plus petit. Si l'inverse est vérifié, i2 est plus petit. Cela correspond au troisième cas de l'algorithme et nous savons maintenant que si nous n'avons pas encore renvoyé de résultat jusqu'à présent, c'est parce que les bornes max sont égales

- Puis on compare ensuite les ouvertures des intervalles. Si la borne max de i2 est ouverte alors que celle de i1 est fermée, i1 est plus petit. Si l'inverse est vérifié, i2 est plus petit. Cela correspond bien au 4eme cas de l'algorithme, et on sait que i1 = i2.

- Ainsi, il ne reste plus que le cas de l'égalité : on renvoie 0 dans tous les autres cas !

Ainsi, notre fonction renvoie bien le bon résultat de la comparaison entre i1 et i2, peu importe la valeur de i1 ou de i2.

Preuve de random_liste_inter :

On se trouve dans la fonction auxiliaire de random_liste_inter :
Pour tout k € N, P(k) : « La liste renvoyée contient bien k intervalles aléatoires en plus de ceux de l'accumulateur »

Base : k = 0
On voit que si k = 0, notre fonction renvoie directement l'accumulateur. Or celui-ci de base est vide. On renvoie donc une liste vide, donc P(0) est vraie.

Récurrence : Supposons n € N tel que P(n) est vraie. Montrons que P(n+1) est vraie également.

On fait notre appel à la fonction avec comme valeur pour k, n+1. Alors comme n+1 est différent de 0 (car n >= 0), on rentre dans l'appel récursif avec aux (n) (O.random_intervalle 50 :: acc)
Or, on sait que P(n) est vérifiée donc cet appel récursif renvoie bien l'accumulateur avec n intervalles aléatoires en plus. Or, comme on a rajouté un intervalle aléatoire, celle-ci contient bien n+1 intervalles aléatoires. Donc P(n+1) est vérifiée.

Donc, par récurrence, pour tout k € N, P(k) est vérifiée. Donc notre fonction se termine bien et renvoie le bon résultat.
Donc random_liste_inter renvoie bien la liste de k intervalles aléatoires (car l'accumulateur de base est vide).

Preuve de affiche_list_inter :

Pour tout k € N, P(k) : « La fonction affiche bien la liste d'intervalle l avec card(l) = k »

Base : k=0
Si k = 0, alors la liste est vide. Donc on ne doit rien afficher. On voit que c'est bien le comportement de la fonction si la liste est vide, donc P(0) est vérifiée.

Récurrence : Supposons n€N tel que P(n) est vraie. Montrons que P(n+1) est vraie également.

On fait l'appel de affiche_list_inter l avec card(l) = n+1.
Alors la liste est non vide donc on affiche le premier intervalle de la liste (on suppose string_of_inter fonctionnel), puis on réalise un appel récursif sur l sans son premier élément (qu'on appelle ts). Alors card(ts) = n. D'après notre hypothèse, cet appel va donc bien se terminer et afficher tous les éléments de la liste. Donc P(n+1) est vérifiée également.

Par récurrence, pour tout n € N, P(n) est vérifiée et donc affiche_list_inter affiche bien tous les intervalles de la liste.

Preuve de liste_vers_interAb :

Notre fonction part d'un arbre vide et ajoute un à un les éléments de la liste en utilisant la fonction List.fold_left. Comme on sait que List.fold_left marche, et que l'ajout marche également, notre fonction renvoie bien un arbre à qui on a ajouté les intervalles de la liste.

Preuve de to_string :

Pour tout n € N, P(n) : « La fonction renvoie bien l'arbre a avec card(a) = n sous forme de String »

Base : n = 0
Alors l'arbre est vide, on voit que la fonction renvoie « Vide » donc P(0) est vérifiée

Récurrence : Supposons n € N* tel que pour tout m<n, P(m) est vérifiée. Montrons que P(n) est vérifiée également

Soit a un arbre avec card(a) = n, donc a différent du Vide.
L'appel à la fonction to_string va donc directement l'envoyer dans le deuxième cas.
On sait que son élément va être bien affiché car string_of_inter fonctionne. Ensuite, il y a deux appels récursifs sur les arbres gauches et droits de a. Or, par définitions des arbres, card(ag) < n et card(ad) < n. Donc d'après l'hypothèse, ils seront bien affichés.
Donc P(n) est vérifiée.

Donc par récurrence généralisée, pour tout n € N, P(n) est vérifiée et donc la fonction renvoie bien l'arbre sous forme de string.

Preuve de is_verified_all :

P(k) : “La fonction renvoie le bon résultat et se termine bien pour toute liste l vérifiant card(l) = k ”

Base : k = 0
Si k = 0, alors la liste l est vide et donc on renvoie l’accumulateur qui de base vaut true. Il s’agit du bon résultat donc P(0) est vérifiée

Récurrence : Supposons n € n tel que P(n) est vraie. Montrons que P(n+1) est vraie également.

Soit l une liste tel que card(l) = n+1. Alors l est différent de la liste vide, et donc on rentre dans le deuxième cas : l est divisé sous forme de queue :: tète. On appelle la queue t et la tête ts. On réalise donc un appel récursif, en testant notre fonction f sur l’élément t et l’arbre a et en ajoutant le résultat à l’accumulateur (si jamais cela renvoie false, l’accumulateur devient false sinon il reste à true). Puis pour l’appel récursif, on donne la liste ts.

Or, card(ts) = n. Par hypothèse de récurrence, on peut donc dire que l’appel récursif se termine et renvoie le bon résultat. Comme on a ajouté notre test sur l’élément t dans l’accumulateur, on peut dire que l’appel se termine et renvoie le bon résultat pour l une liste vérifiant card(l) = n+1.

Par récurrence, pour tout k€N, P(k) est donc vraie.
Donc la fonction renvoie bien le bon résultat.

Preuve de valeur_inter_dans_arbre :

On prend en compte l'ordre >E qui est bien fondé. Soit E l'ensemble des éléments de l'intervalle.

Propriété : « valeur_inter_dans_arbre se termine et renvoie le bon résultat »

Cas de base : On se place dans la fonction auxiliaire et on prend min et max tel que min > max. Alors on renvoie directement l'accumulateur de base qui vaut true. Cet intervalle imaginaire ne peut pas exister mais il sert de cas d'arrêt pour tous les intervalles, et renvoie donc bien le bon résultat.

Hypothèse d'induction : Soit m € E tel que pour tout n >E m, la propriété est vérifiée pour n et tel que m <= max.

Alors comme compElement m max <= 0, on rentre dans le if et on fait un appel récursif. L'appel récursif se fait sur le prochain élément de m qu'on appelle m'. Or, m' >e m donc la propriété est vérifiée pour m'. Donc l'appel récursif se termine et renvoie le bon résultat pour m'. De plus, on a rajouté à l'accumulateur la présence de m ou non dans a. Ainsi, la propriété est également vérifiée pour m.

Par induction, on peut donc dire que la propriété est vérifiée pour tout m€E, et donc pour tout intervalle et pour tout arbre.

Preuve de liste_disjoints :

L'algorithme de liste_disjoints se base sur le fait que la liste l donné en paramètre est supposée triée. Ainsi, il suffit de comparer un élément avec son suivant sur toute la liste pour savoir si tous les éléments de la liste sont disjoints deux à deux.
En effet, on prend une liste de 3éléments triés de tel sorte que le premier élément est le plus petit et le dernier le plus grand. Si le premier élément est disjoint avec le deuxième, alors comme le troisième élément est plus grand ou égale au second, on sait que le premier élément sera également disjoint avec le troisième (la borne min du deuxième élément est plus grande que la borne max du premier élément, donc comme la borne min du troisième élément est plus grande ou égale que celle du deuxième, ils sont disjoints)

Pour tout k € N, P(k) : « les éléments de la liste l sont deux à deux disjoints avec card(l) = k et la fonction renvoie le bon résultat »

Base : k = 0
Alors la liste est vide et on renvoie true, c'est le bon résultat.

Récurrence : Soit n € N* tel que P(n) est vraie. Montrons que P(n+1) est vraie également.

Soit l une liste d'intervalle tel que card(l) = n + 1

Alors la liste est non vide, donc on rentre dans le deuxième cas de notre fonction.
La liste l est divisée en t :: ts et un appel récursif est fait sur ts. De plus, le test de disjonction entre t et l'ancien élément previousI est rajouté à l'accumulateur. Or, card(ts) = n donc par hypothèse de récurrence, l'appel récursif se termine bien et renvoie le bon résultat. Or, comme on a rajouté notre test sur t à l'accumulateur, on renvoie bien le bon résultat pour card(l) = n+1. Donc P(n+1) est vérifiée.

Par récurrence, la propriété P est donc validée pour tout k € N.
Donc liste_disjoints renvoie bien le bon résultat.

Preuve de est_dans_arbre_retrait :

Pour tout n € N, P(n) : « La fonction se termine bien et renvoie le bon résultat pour card(l) = n »

Base : n = 0
Alors la liste est vide, la fonction renvoie donc directement l'accumulateur qui vaut true de base. Il s'agit du bon résultat car rien est bien inclut dans a' et n'est pas dans i' (par défaut) donc P(0) est vérifiée.

Récurrence : Soit n € N tel que P(n) est vérifiée. Montrons que P(n+1) est vérifiée également.

Soit l une liste d'intervalle avec card(l) = n+1
Alors la liste est différente de l'arbre vide et se décompose en t :: ts
Il y a deux cas différents :

- Si t se trouve dans a', alors on teste s'il se trouve dans i' ou non et on rajoute ça à notre accumulateur.
- Sinon, on ne l'ajoute pas à l'accumulateur, pas de test en plus

Dans les deux cas, on réalise ensuite un appel récursif sur ts. Or, card(ts) = n donc d'après l'hypothèse de récurrence, l'appel se termine et renvoie bien le bon résultat. Comme on a bien ajouté notre test à l'accumulateur si nécessaire, on renvoie bien le bon résultat et donc P(n+1) est vérifiée.

Par récurrence, pour tout n € N, P(n) est vérifiée et donc la fonction se termine bien et renvoie le bon résultat.

Preuve de valide_arbre_inter :

Nous avons réalisé toutes les preuves de toutes les fonctions appelées dans valide_arbre_inter auparavant. Or, comme cette fonction ne fait qu'appliquer l'algorithme du sujet, celle-ci réalise bien ce qui est demandé et se termine bien.

Preuve de union :

Pour réaliser l'union, on se base sur l'algorithme suivant :

- L'union d'un intervalle avec le vide donne ce même intervalle
- L'union de deux intervalles collés avec des bornes opposées (en termes d'ouverture) au niveau de la borne en commun donne l'intervalle composé des bornes mins du plus petit et des bornes max du plus grand
- L'union de deux intervalles non disjoints renvoie l'intervalle composé de la borne min du plus petit et de la borne max du plus grand si la borne min du plus petit est inclus dans le plus grand, sinon la borne max du plus petit
- L'union de deux intervalles disjoints i1 et i2 donne i1 U i2 (on ne peut pas les fusionner) si jamais ils ne sont pas collés au niveau d'une borne avec des ouvertures différentes.

Cet algorithme permet de traiter tous les cas possibles et réalise bien une union d'intervalle.

Prouvons que notre fonction réalise bien notre algorithme :

- Les deux premiers cas expliquent qu'une union d'intervalle avec le vide renvoie ce même intervalle. Il s'agit bien du premier cas.
- Le troisième et le quatrième cas expliquent que l'union de deux intervalles collés avec des bornes opposés au niveau de la borne en commun renvoi l'intervalle composé de la borne min du plus petit et de la borne max du plus grand. Cela correspond bien au deuxième cas.
- Le cinquième cas, le plus long, traite les cas d'intervalle non disjoints. Il va récupérer la borne min du plus petit et la borne max du plus grand si la borne max du plus petit est inclut dans le plus grand, la borne max du plus petit sinon. Cela réalise bien le troisième cas.
- Le dernier cas traite les intervalles disjoints. Celui-ci renvoie un couple composé des deux intervalles. Or, on sait que les intervalles ne peuvent pas être collés au niveau d'une borne en ayant des ouvertures différentes (il s'agit des cas 3 et 4 qui ont déjà été traité). Ainsi, on sait donc que cela réalise le quatrième cas.

Notre fonction réalise donc bien l'algorithme d'union, celle-ci renvoie le bon résultat.

Preuve de différence :

Pour réaliser la différence, on se base sur l'algorithme suivant :

- Un intervalle moins le vide renvoie cet intervalle
- Le vide moins un intervalle renvoie le vide
- La différence de deux intervalles disjoints renvoie le premier intervalle
- La différence entre un intervalle i1 et i2 avec i1 inclut dans i2 donne le Vide
- La différence entre un intervalle i1 et i2 avec bornemin i1 inclus dans i2 (sans prendre en compte les ouvertures) et bornemax i1 > bornemax i2 donne l'intervalle avec comme borne min la borne max de i2, comme borne max la borne max de i1 et comme ouverture à gauche l'opposé de l'ouverture de la borne max de i2 et comme ouverture à droite l'ouverture de la borne max de i1
- La différence entre un intervalle i1 et i2 avec bornemax i1 inclus dans i2 (sans prendre en compte les ouvertures) et bornemin i1 < bornemin i2 donne l'intervalle avec comme borne min la borne min de i, comme borne max la borne min de i2 et comme ouverture à droite l'opposé de l'ouverture de la borne min de i2 et comme ouverture à gauche l'ouverture de la borne min de i1
- [x ;y] –]x ;y[= [x ;x] U [y ;y]
- [x ;y] – [x ;y[= [y ;y]
- [x ;y] -]x ;y] = [x ;x]
- La différence entre un intervalle i1 et i2 quand les bornes mins sont égales avec i1 fermé à gauche et i2 ouvert à gauche et bornemax i1 > bornemax i2 donne l'intervalle fermé composé de seulement la borne min de i1 à gauche et à droite union l'intervalle composé de la borne max de i2 à gauche, de la borne max de i1 à droite, avec l'ouverture opposée de la borne max de i2 à gauche, et l'ouverture de la borne max de i1 à droite
- La différence entre un intervalle i1 et i2 quand les bornes max sont égales avec i1 ouvert à droite et i2 fermé à droite et bornemin i2 > bornemin i1 donne l'intervalle composé de la borne min de i1 à gauche, de la borne min de i2 à droite, avec la même ouverture à gauche que l'ouverture de i1 à gauche et l'ouverture opposée à droite à celle de i2 à gauche union l'intervalle fermé composé de la bornemax de i1 à gauche et à droite
- La différence entre un intervalle i1 et i2 quand la borne min de i2 est supérieur à la borne min de i1 et que la borne max de i2 est inférieur à la borne max de i1 donne l'intervalle composé de la borne min de i1 à gauche avec son ouverture et de la borne min de i2 à droite avec son ouverture opposée union l'intervalle composé de la borne max de i2 à gauche avec son ouverture opposée et de la borne max de i à droite avec son ouverture.
- Dans tous les autres cas, on renvoie i1

On sait que les intervalles renvoyés par l'algorithme sont corrects du fait des préconditions posées sur les valeurs de i1 et de i2. De plus, on voit que tous les cas sont traités. L'algorithme est un peu laborieux mais se simplifie pas mal dans le code, étant donné que chaque bornemin/bornemax ne représente plus qu'une ou deux lettres/chiffres.

Prouvons que notre fonction réalise bien notre algorithme :

- Dans le cas où i2 vaut vide, on renvoie i1. Cela correspond bien au premier cas de l'algorithme.
- Dans le cas où i1 vaut le vide, peu importe la valeur de i2, on renvoie le vide. Cela correspond bien au deuxième cas de l'algorithme
- Lorsque i1 et i2 sont disjoints, on renvoie i1. Cela correspond bien au troisième cas de l'algorithme.
- Le test du quatrième cas correspond au test de l'inclusion. Si i1 est inclus dans i2, on renvoie le vide. Cela correspond bien au quatrième cas.
- Le cinquième et le sixième cas réalisent bien ce que demande l'algorithme dans les cas 5 et 6, et renvoient bien le bon résultat. Il s'agit juste d'une version « formelle » de l'algorithme, les tests dans le when sont la version mathématique de ce qui est dit dans l'algorithme.
- Les cas 7, 8 et 9 correspondent bien au cas 7 8 et 9 de l'algorithme. On voit que les préconditions sur i1 et i2 font qu'ils correspondent exactement aux intervalles de l'algorithme, et on renvoie bien un intervalle similaire à ceux de l'algo.
- Enfin, tous comme les cas 5 et 6, les cas 10, 11, et 12 sont une version formelle des cas de l'algorithme.
- Finalement, le dernier cas renvoie bien i1 peu importe ce qu'il y à gauche, comme prévu.

Notre fonction réalise donc bien l'algorithme de la différence, et elle renvoie donc le bon résultat.

Preuve de est_dans_arbre_intervalle :

L'algorithme de est_dans_arbre_intervalle se base sur la logique suivante :

- L'élément se trouve dans l'intervalle du nœud, dans ce cas-là c'est vrai
- L'élément est plus petit que la borne min de l'intervalle, dans ce cas-là on fait un appel récursif sur l'arbre gauche
- L'élément est plus grand que la borne max de l'intervalle, on fait un appel récursif sur l'arbre droit
- Sinon, l'intervalle vaut le Vide et donc on renvoie faux

Prouvons que notre fonction réalise bien cet algorithme :

- Si notre élément se trouve directement dans l'intervalle du nœud, on renvoie true. Il s'agit bien du premier cas.
- S'il est plus petit que la bornemin de l'intervalle, alors on fait un appel récursif sur l'arbre gauche. Il s'agit bien du deuxième cas.
- Sinon, si l'arbre n'est pas vide (il se décompose en Inter ()), on fait un appel récursif sur l'arbre droit. On sait que cela correspond au troisième cas, car l'élément n'étant pas dans l'intervalle, et n'étant pas plus petit que la borne min, il est forcément plus grand que la borne max.
- Enfin, dans tous les autres cas, on renvoie faux. Cela correspond bien au cas du vide, donc au quatrième cas.

Notre fonction réalise donc bien l'algorithme, et renvoie donc le bon résultat.

Preuve de random_intervalle :

Pour réaliser un intervalle aléatoire, on prend deux entiers aléatoires. Ces entiers sont placés dans les valeurs min et max, en fonction de qui est le plus petit. Ainsi, on sait que min <= max. De plus, si min = max, on force les bornes à être fermées, sinon on met des ouvertures aléatoires. Ainsi, on sait que notre intervalle est bien valide, et que l'intervalle est aléatoire.

Preuve de is_included :

Pour connaitre l'inclusion, on se base sur l'algorithme suivant : Un intervalle est inclus dans un autre si leur différence vaut le vide.

Or, on sait que notre fonction différence est valide, on compare le résultat de la différence avec le Vide et on renvoie ce résultat. Ainsi, notre fonction inclusion est valide et renvoie le bon résultat.

Preuve de is_included_union :

Pour savoir si i1 est inclut dans i2 union i3, on regarde tout simplement si i1 est inclut dans i2 ou s'il est inclus dans i3. En effet, on sait que i1 ne peut pas être un couple, et que i2 et i3 sont disjoints, donc il est inclus soit dans l'un, soit dans l'autre.

Preuve de valide_diff :

Pour réaliser valide_diff, on se base sur l'algorithme suivant : pour savoir si les valeurs de i1 sont exclusivement dans i2 ou dans i4, on regarde si i4 est inclus dans i1, et i4 et i2 sont disjoints, et si les valeurs de i1 moins celle d'i4 sont dans i2.

Notre fonction réalise exactement cela, en prenant en compte le fait que i4 peut être un couple. Il faut donc potentiellement réaliser la différence deux fois avec i1, pour chaque valeur du couple.
On sait que notre fonction différence est valide, tout comme is_included_either_2 et sont_disjoints, donc notre fonction renvoie bien le bon résultat.

Preuve de valide_inter :

Ainsi, toutes les fonctions utilisées dans valide_inter sont valides. Celle-ci ne faisant qu'appliquer l'algorithme du sujet, elle est donc valide.