ENS Lyon

Typage de protocoles objet en Mezzo

Stage à l'équipe Gallium (Inria)

Auteur :
Armaël Guéneau

Maître de stage :
François Pottier

25 août 2013

Table des matières

Introduction

Dans le cadre du « stage d'initiation à la recherche » se déroulant en fin d'année de L3, j'ai choisi de passer les 7 semaines de celui-ci au sein de l'équipe-projet **Gallium** du centre de recherche Inria Paris-Rocquencourt.

L'équipe **Gallium**, descendante de l'équipe **Cristal**, s'intéresse à la façon dont les programmes informatiques sont exprimés par les humains, jusqu'à leur traduction en langage machine. Ses recherches portent donc sur la conception, la formalisation et l'implémentation de langages de programmation, et visent à accroître la fiabilité des logiciels via des langages et outils plus aptes à garantir leur sûreté de fonctionnement.

Le compilateur CompCert, conjointement écrit et prouvé en Coq, est par exemple une réussite dans ce domaine. On peut également citer OCaml, lui aussi développé à **Gallium**, un langage de programmation de haut niveau fortement et statiquement typé.

Développé par mon maître de stage François Pottier, son doctorant Jonathan Protzenko et le post-doctorant Thibaut Balabonski, le langage Mezzo constitue une autre avancée dans cette voie.

Mon stage consistait en l'utilisation de ce langage. Plus précisément, il s'est intéressé à la question de savoir comment exprimer des *protocoles objet* grâce au système de type élaboré de Mezzo, afin de garantir statiquement leur respect par l'utilisateur.

Mon travail a donc consisté à écrire du code Mezzo pour essayer d'exprimer ces propriétés, la réalisation principale étant l'implémentation d'itérateurs, que l'on retrouve actuellement dans la bibliothèque standard de Mezzo.

L'utilisation de Mezzo a également motivé l'ajout de fonctionnalités au typeur, lorsqu'elles se sont révélées nécessaires.

Je tiens à remercier les chercheurs, permanents et doctorants de Gallium pour l'excellente ambiance de l'équipe. Je remercie François Pottier pour m'avoir dirigé, guidé, et avoir été disponible dès que j'en avais besoin, tout comme Jonathan Protzenko et Thibaut Balabonski. Merci à Gabriel Scherer pour m'avoir encouragé à écrire un billet de blog. Merci à tous pour l'accueil durant cette période qui fut fort enrichissante.

1 Un nouveau langage de programmation : le cas Mezzo

Mezzo [10] est un langage de programmation de haut niveau, inspiré par ML. Ce qui le distingue, et qui nous intéresse ici, est son système de type particulier. Comparé aux variantes de ML comme OCaml, l'objectif est double :

- *Étendre* leur système de types, ce qui signifie pouvoir typer plus de programmes
- Le *raffiner*, afin de fournir des garanties plus fortes

Il s'agit donc de pouvoir raisonner plus finement sur un plus grand nombre de programmes.

1.1 Objectifs

L'idée derrière Mezzo est d'affiner le système de types de ML, afin de pouvoir parler d'un certain état des objets manipulés. C'est en effet une notion qui revient fréquemment dans de nombreux programmes. Un fichier sera par exemple *ouvert*, puis passera dans l'état *fermé*, état dans lequel il n'est plus possible de lire ou écrire, ou de le (re)fermer.

Cette idée, appliquée naïvement, n'est pas satisfaisante. En effet, certains programmes ne respectant pas le protocole seraient acceptés, alors qu'ils produiraient une erreur. Le code suivant en est un exemple : il montre la nécessité de contrôler - dans le système de types - la manière dont sont désignés les objets en mémoire.

```
let f = file_open "fichier.txt" in
(* f est ouvert *)
let f' = f in
(* f est ouvert, f' est ouvert *)
...
(* f est ouvert, f' est ouvert *)
close f;
(* f est fermé, f' est ouvert *)
close f';
(* Crash! *)
```

Ce programme est considéré comme *faux* par Mezzo, qui par conséquent le rejette. Pour cela, le système de types de Mezzo, à la place de « `f' est ouvert` », infère « `f' = f` » : lors de l'appel `close f'`, Mezzo suit l'égalité, et obtient donc que « `f' = f est fermé` », ce qui contredit le protocole.

Mezzo fournit ainsi un contrôle fin sur le nommage d'un même objet mémoire à différents endroits (*aliasing*), et l'appartenance de l'objet mémoire à une section de programme (*ownership*).

Un autre objectif de Mezzo est, malgré son système de types sophistiqué, de garder un certain degré d'automatisation : on ne veut pas avoir de preuves à écrire à la main pour typer nos programmes. Un certain nombre d'annotations est cependant nécessaire [1], comme par exemple pour le type des fonctions.

1. Il n'y a pas de types principaux en Mezzo : pour un terme donné, il n'existe pas un type pour ce terme tel que tous les autres types du terme en sont une instance. Contrairement à ML où l'algorithme d'inférence recherche le type principal pour un terme, le typeur doit ici faire un choix entre plusieurs types qui ne sont pas équivalents, nécessitant parfois une annotation du programmeur

1.2 Le système de permissions de Mezzo

Mezzo introduit la notion de *permission*, similaire à une assertion en logique de séparation [11]. Les permissions n'existent pas lors de l'exécution du programme. Un terme `x` n'a pas un type fixé : à la place, il est possible de posséder, à un certain point, une permission `(x @ t)`, autorisant à utiliser `x` comme ayant le type `t`. Cette permission peut être transférée, disparaître, ou être remplacée par une autre.

Par ailleurs, dans le cas où `x` pointe sur un objet mutable, il est assuré qu'il n'y aura toujours qu'une seule permission autorisant à lire et écrire dans `x`. Cette permission est *exclusive*, on ne peut la dupliquer. À l'inverse, pour un objet immutable, la permission autorisant à lire l'objet est *duplicable*, et sera donc présente en autant d'exemplaires que nécessaire.

```
let x = 3 in
(* (x @ int) : duplicable *)
let y = x in
(* x @ int * y = x *)
assert (x @ int * y @ int)
(* Ok : la permission sur x
   est dupliquée *)
```

Listing 1 : x est immutable

```
let x = newref 3 in
(* (x @ ref int) : exclusive *)
x  := 2;
(* x @ ref int *)
let y = x in
(* x @ ref int * y = x *)
assert (x @ int); assert (y @ int);
(* Ok : on a bien une permission *)
assert (x @ int * y @ int)
(* Mais pas deux : erreur de typage *)
```

Listing 2 : x est mutable

Cette notion nous permet bien d'exprimer l'état d'un objet : un changement d'état est représenté par un changement de type, où `(x @ t)` deviendrait par exemple `(x @ t')`. Puisque les objets mutables ont, à tout moment, un unique possesseur, il est sûr de changer le type d'un objet.

Le fait qu'une seule partie du programme puisse avoir accès à un objet mutable à la fois, du fait de l'exclusivité de la permission sur cet objet, est également utile dans le cas d'un programme concurrent. Ainsi, le système de types de Mezzo garantit statiquement qu'un programme ne peut contenir de situations d'accès concurrent à une même donnée (*data race*).

> **Point Mezzo 1** *Je ne vais pas introduire d'une seule traite la syntaxe de Mezzo. Au contraire, j'introduirai les notions au fur et à mesure que cela sera nécessaire pour comprendre les morceaux de code exposés, dans des cadres semblables à l'actuel.*

1.3 Visées du stage

Mon travail au long de ce stage concerne la notion de *protocole objet* [12]. Étant donné un objet pouvant se trouver dans plusieurs états, et des fonctions le faisant passer d'un état à un autre, un protocole objet est une suite de règles décrivant dans quel ordre il est légal d'appliquer ces fonctions, selon l'état dans lequel se trouve l'objet. Un exemple est celui de l'objet `fichier` : il est légal d'appeler `read`/`write` uniquement après l'avoir ouvert, ainsi que `close` qui change l'état du fichier d'« ouvert » à « fermé ».
Afin de garantir de telles propriétés, il est possible de les exprimer dans le système de types : on parle alors parfois d'analyse de *typestate* ("typestate analysis").

L'intérêt pratique est de pouvoir formaliser des conventions d'une API ou bibliothèque. De nombreuses bibliothèques comportent en effet des protocoles implicites, qui ne sont, la plupart du temps, pas exprimés dans le langage de programmation lui même. À la place, il est généralement indiqué dans la documentation que si l'on appelle telle fonction après telle autre, il se produira une erreur à l'exécution, ou pire, que le comportement est alors indéfini.
À l'inverse, si le protocole est exprimé dans le système de type, avec par exemple un type différent par état, celui-ci permettra alors de garantir statiquement (à la compilation) que le protocole ne sera pas violé.

Comme affirmé précédemment, Mezzo semble a priori particulièrement adapté pour cette tâche, puisqu'une permission représente justement l'état d'un objet en mémoire, et de ce qu'il est possible de faire avec. Il va donc être question de mettre cette affirmation à l'épreuve de protocoles et implémentations un peu évoluées.

Le problème est double : il s'agit d'une part, pour un protocole donné, de déterminer dans quelle mesure il est possible de l'axiomatiser dans le système de types, à savoir, écrire des types et des *signatures* de fonctions qui le représentent. D'autre part, il s'agit de savoir s'il est possible d'*implémenter* effectivement les fonctions pour les signatures données.

La section 2 concerne le premier problème. Pour une implémentation existante (non sûre) donnée, j'étudie la manière dont il est possible d'écrire une interface sûre en Mezzo, garantissant une utilisation conforme au protocole. La méthode permettant d'obtenir cette interface ne s'applique qu'à une certaine classe de protocoles, que je caractérise.

La section 3 s'intéresse quant à elle à l'implémentation réelle, en Mezzo, d'un protocole simple. Ce n'est pas aussi simple qu'il n'y paraît au premier abord, et permet de mettre en évidence des *design patterns* propres à Mezzo, mais aussi les difficultés inhérentes à un langage tenant compte de l'*ownership* des objets mémoire.

2 Axiomatisation d'APIs Posix grâce au système de types

On s'intéresse donc ici à l'écriture d'une interface en Mezzo pour des APIs Posix existantes. On se concentre ici tout d'abord sur un sous ensemble de l'API permettant de lire et écrire dans des fichiers, puis sur l'API *Sockets*, qui suit un protocole plus compliqué.

2.1 API *Fichiers*

L'API Posix de manipulation des fichiers suit un protocole plutôt simple, qui nous servira d'échauffement. On se restreint au cas où l'on utilise les fonctions `fopen`, `fread`, `fwrite` et `fclose`.

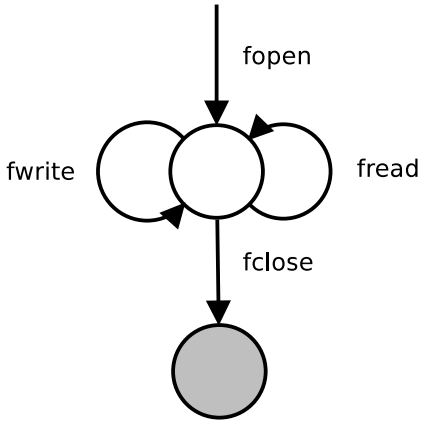

FIGURE 1 – Protocole pour notre sous-ensemble de l'API Fichiers [7]

On définit donc un type abstrait correspondant à la structure `FILE*` sous-jacente. Cette structure étant mutable, on déclare le type comme étant exclusif : les permissions pour ce type le seront également.

On définit également un type abstrait représentant un fichier fermé après appel à `fclose()` [2].

```
abstract file

abstract closed_file
```

Les signatures des fonctions `fopen`, `fread` et `fwrite` s'écrivent sans difficulté, en se calquant sur le prototype des fonctions C sous-jacentes.

```
val fopen : (string, string) -> file

val fread : (file, string, int, int) -> int
val fwrite : (file, string, int, int) -> int
```

Les choses intéressantes se passent au niveau de la fonction `fclose`.

2. Ce type n'est en réalité pas vraiment utile, puisqu'on ne peut le passer à aucune fonction. Il sert ici à illustrer le changement d'état via un changement de type - en pratique on préférera remplacer une permission (`x @ closed_file`) par une permission vide (`empty`).

> **Point Mezzo 2** *En Mezzo, si une fonction* `f` *a le type* `t1 -> t2`*, cela signifie que lors de l'appel* `let y = f x`*, la permission* `(x @ t1)` *doit être présente dans l'environnement. Après l'exécution de la fonction,* `(x @ t1)` *est toujours présente, et vient s'ajouter* `(y @ t2)`*.*
>
> *Lorsque le type de* `f` *est* `(consumes t1) -> t2`*,* `(x @ t1)` *doit toujours être présente avant l'exécution de* `f`*, mais est consommée par* `f`*. Seule la permission* `(y @ t2)` *est présente après l'exécution de* `f`*.*

> **Point Mezzo 3** *Un type en Mezzo peut être formé en associant une permission et un type préexistant : si* `t` *est un type,* `p` *une permission, alors* `(t | p)` *est un type, signifiant « le type* `t` *sachant que l'on a la permission* `p` *». Dans le cas où* `t` *est* `()` *("unit", le type a un seul habitant), on peut écrire* `(| p)`*.*

```
val fclose : (consumes f : file) -> (| f @ closed_file)
```

On remarque qu'il est possible de nommer les arguments d'une fonction, même dans la signature, et se servir du nom dans le type de retour.

`fclose` exprime effectivement le changement d'état : si la permission `(f @ file)` est présente dans l'environnement, il la consomme et la remplace par `(f @ closed_file)`. Il est alors impossible de fermer deux fois par mégarde le même fichier :

```
let f = fopen("fichier.txt", "r") in
(* f @ file *)
let _ = fread(f, ...) in
(* f @ file *)
fclose(f);
(* f @ closed_file *)
...
fclose(f);
(* Erreur de typage :
   (f @ file) n'est pas dans l'environnement *)
```

Bilan On voit ici une première méthode permettant d'exprimer un protocole en Mezzo. Elle utilise directement les primitives fournies par le langage, en définissant un type par état, les fonctions/transitions consommant la permission associée, et générant une permission associée au nouvel état.

2.2 API *Sockets*

Cette méthode, telle quelle, a cependant des limites. Le protocole représenté figure 2 pose par exemple problème : les états `a` et `b` étant représentés par des types différents, il faudrait pouvoir exprimer que `f` prend en entrée une valeur de type `b`, ou de type `c`, ce qui n'est pas possible.

Le protocole suivi par les sockets Posix (`socket.h`) comporte lui aussi des complications de ce genre. Il nous faut donc une nouvelle méthode, moins restreinte.

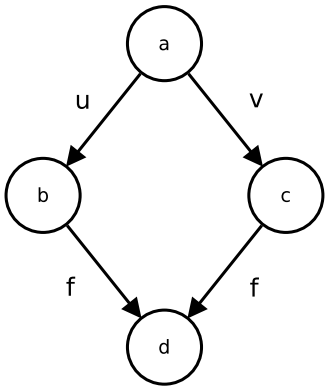

FIGURE 2 – Protocole non exprimable avec la méthode de la section 2.1

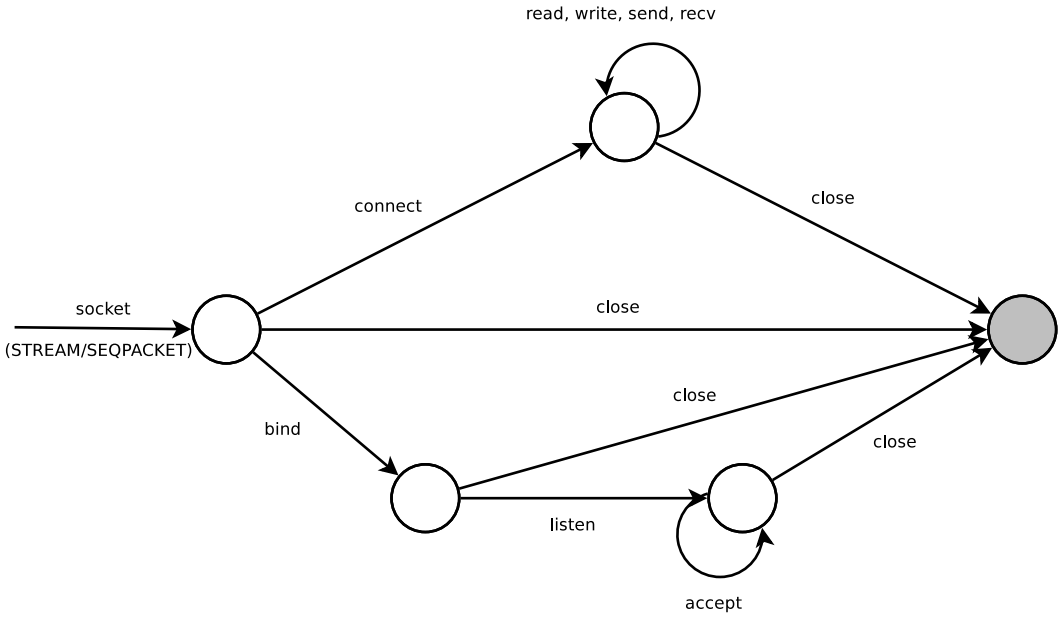

FIGURE 3 – Protocole pour un socket créé en mode connecté [7]

L'idée, inspirée par [1], est d'utiliser des permissions "fantômes" en guise de prédicats, exprimant la possibilité d'appeler telle ou telle fonction.

Ce que j'appelle permission fantôme est une permission abstraite, utile seulement pour l'interface, et dont l'implémentation effective est la permission `empty`. Dans notre cas, à chaque fonction (`u`, `v`, `f`) correspond une permission fantôme (`can_u`, `can_v`, `can_f`) paramétrée par un terme, signifiant qu'il est possible, pour ce terme, d'appeler la fonction correspondante.

Un état consiste alors en la collection des permissions-prédicats correspondant aux fonctions qu'il est possible d'appeler. Lors de l'appel d'une fonction `u` avec l'argument `x`, celle-ci consomme la permission `can_u x` : plus besoin de savoir exactement dans quel état on se trouve. On résout ainsi le problème de disjonction : un état sera codé par une conjonction de permissions `can_*`, et il sera possible d'oublier si besoin une partie de ces permissions pour extraire la partie commune qui nous intéresse. Cela se fait de

manière transparente, sans annotations de types.

Puisque l'on ne peut plus associer un type à chaque état du protocole, on considérera que l'objet suivant le protocole conserve un même type `t`, qui serait par exemple le type `socket` dans le cas du protocole homonyme.

```
abstract can_u (x : term): perm
abstract can_f (x : term): perm
val u : (x : t | consumes can_u x) -> (| can_f x)
```

> **Point Mezzo 4** *Il y a trois sortes (kind) en Mezzo :* `term`, `type` *et* `perm` *(désignant respectivement les termes, types et permissions). Il est ainsi possible, lors d'une déclaration avec* `abstract` *d'annoter les paramètres et la valeur déclarée par la sorte voulue. De manière générale, si une annotation est absente, la sorte* `type` *est choisie par défaut.*

Un problème se pose, cependant : il reste dans l'environnement des permissions résiduelles, correspondant à d'autres fonctions que l'on aurait pu appeler. Après changement d'état, ces permissions ne sont plus valides, pourtant elles peuvent toujours être utilisées.

```
(* x est dans l'état a *)
(* x @ t * can_u x * can_v x *)
u x;
(* x @ t * can_f x * can_v x : permission résiduelle *)
v x;
(* Crash ! *)
```

> **Point Mezzo 5** *Pour* `p` *et* `q` *deux permissions, la permission* `p * q` *correspond à la possession d'à la fois* `p` *et* `q`

On paramètre alors le type `t` de `x`, et les prédicats par une étiquette (*stamp*) unique, modifié à chaque changement d'état. Les anciens prédicats seront alors inutilisables, puisque paramétrés par un stamp différent de celui du type `t` courant.

Il n'est pas nécessaire d'expliciter ce stamp : il suffit de le quantifier existentiellement.

> **Point Mezzo 6** *Il est possible en Mezzo de quantifier existentiellement et universellement sur un type, terme ou permission. La syntaxe correspondant à* $\exists x : \kappa.t$ *(où* κ *est une indication de kind) est* `{ x : κ } t`*. La syntaxe pour* $\forall x : \kappa, t$ *est* `[x : κ] t`

```
abstract t (stamp : type)

abstract can_u (x : term) (stamp : type): perm
```

```
abstract can_v (x : term) (stamp : type): perm
...

(* États *)
alias a : type = { stamp } (x : t stamp | can_u x stamp * can_v x stamp)
alias b : type = { stamp } (x : t stamp | can_f x stamp)
...

val u : [stamp] (x : t stamp | can_u x stamp) -> (| x @ b)
...
```

> **Point Mezzo 7** *Il est utile de connaître le comportement du typeur vis à vis des types quantifiés existentiellement.*
>
> *Lors de l'apparition dans l'environnement d'un type quantifié existentiellement (*`{x:κ} y`*), le typeur le « dépaquette »* ***immédiatement*** *: il génère un nouveau nom abstrait correspondant au témoin de la quantification (le* `x`*), et ajoute le type* `y` *particularisé pour ce* `x`*.*
>
> *Dans l'autre sens, lorsqu'il est nécessaire, pour un type* `y`*, de l'« empaqueter » en* `{x:κ} y`*, pour satisfaire une signature par exemple, le typeur Mezzo recherchera automatiquement dans l'environnement un témoin* `x` *satisfaisant la propriété, et ajoutera* `{x:κ} y` *à l'environnement.*

Puisque le typeur génère un témoin abstrait à chaque fois qu'un type quantifié existentiellement apparaît, c'est le typeur lui même qui génère des stamps différents : pour chaque état qui apparaît dans l'environnement et dont la quantification est « dépaquetée », un nouveau stamp est créé.

Ce mécanisme garantit ainsi que cette méthode exprime bien le protocole. Elle est par ailleurs moins restreinte que celle vue précédemment, et permet ainsi d'axiomatiser le protocole suivi par les sockets Posix. Le code est disponible en annexe A.1 dans le cas des sockets en mode connecté.

Bilan Bien que plus élaborée, cette méthode couvre une gamme déjà bien plus large de protocoles, offre un premier aperçu de la gestion des existentielles par le typeur.

Limitations Il n'est cependant pas possible d'exprimer un protocole quelconque. En effet, lors d'un changement d'état effectué par une fonction, celle-ci doit désigner le nouvel état *j*, en renvoyant par exemple la permission `(x @ j)`. Le cas où l'état d'arrivée dépend de l'état de départ n'est ainsi pas exprimable. Autrement dit, cette méthode ne permet d'exprimer que les automates dont toutes les transitions avec la même étiquette aboutissent au même état.

Par ailleurs, même si cet encodage utilisant un *stamp* est correct, il est un peu étrange à utiliser, et « polluera » l'environnement de nombreuses permissions `can_*` périmées : cela n'est pas efficace et compliquera les messages d'erreur du typeur. Ce n'est pas très satisfaisant.

3 Implémentation d'un protocole simple en Mezzo : iteration, itérateurs

Ce qui nous a intéressé jusque là était la spécification d'une implémentation écrite dans un langage externe grâce au système de types de Mezzo, afin d'apporter des garanties statiques supplémentaires.

Cependant, Mezzo est avant tout un langage de programmation à part entière : on peut légitimement vouloir implémenter des bibliothèques respectant un protocole objet en Mezzo. C'est relativement différent : il faudra cette fois convaincre Mezzo que l'algorithme lui même respecte le protocole que l'on aura exprimé dans la signature.

Le protocole choisi est celui d'un itérateur : il est (très) simple, et permet par ailleurs d'illustrer comment le transfert d'*ownership* peut s'exprimer en Mezzo.

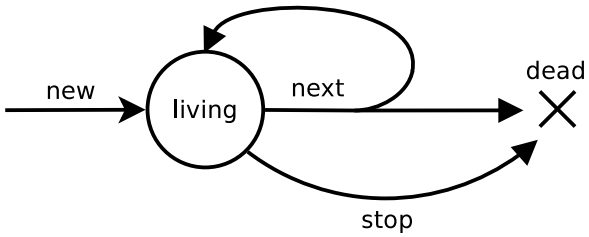

FIGURE 4 – Protocole suivi par un itérateur

3.1 Avant de commencer...

Ce que j'appelle itérateur est un objet permettant d'itérer sur une collection, donnant un nouvel élément à chaque appel à une fonction `next`, qui fait avancer l'itérateur d'une étape en avant. Notons que l'itérateur est mutable, son état interne étant modifié par l'appel à `next`.

Il est nécessaire de gérer le cas où l'itérateur ne contient plus de nouvel élément. En Scala [8] ou Java [1], il faut au préalable vérifier si l'itérateur contient un nouvel élément avec `hasNext`. Appeler `next` sur un itérateur vide lance une exception. De notre côté, nous voulons un encodage statique de la logique de l'itérateur : `next` renverra un type somme à deux constructeurs, indiquant si l'on a pu extraire un élément de celui-ci, ou s'il est vide, auquel cas *il n'est plus possible de l'utiliser.*

3.1.1 Structures de données inductives

L'itération se fait classiquement sur une structure de données. Il nous est facile de définir une structure de liste ou d'arbre grâce aux types de données inductifs de Mezzo.

```
data mutable tree a =
| Leaf
| Node { left : tree a; elem : a; right : tree a }
```

Listing 3 : Un arbre binaire mutable

> **Point Mezzo 8** *Dans* `data tree a = ...`, `a` *est un paramètre de type : la structure est paramétrée par le type de ses éléments.*

Cette définition est similaire à celle qu'on peut trouver en ML. Par ailleurs, posséder la permission `(t @ tree a)` donne accès à `t` en tant qu'arbre, et donc à tous ses éléments et à leur contenu.

Suite à un examen de la structure de `t` (en utilisant le *pattern-matching*, par exemple), cette permission peut être raffinée en `t @ Node { left : tree a; elem : a; right: tree a}`, automatiquement transformée par le typeur en :

```
t @ Node { left = l; elem = x; right = r } *
l @ tree a * x @ a * r @ tree a
```

> **Point Mezzo 9** `left = l` *est du sucre syntaxique pour* `left: (=l)`*, et, pour un terme* `x` *donné,* `(=x)` *est le type des termes égaux à* `x` *(type singleton).*

3.1.2 Itération classique avec fonction d'ordre supérieur

Grâce à ce procédé de « découpe » d'une permission, il est facile d'écrire une fonction explorant récursivement l'arbre. On écrit donc aisément une fonction d'ordre supérieur `iter`, appliquant une fonction passée en argument sur chaque nœud.

```
val iter : [a, s : perm] (
  f : (a | s) -> bool,
  t : tree a
| s) -> bool
```

Listing 4 : Signature de `iter`

La fonction f n'a accès qu'à un élément de l'arbre à la fois : elle reçoit une permission sur un élément de type `a`, et doit la redonner après son exécution : l'élément en question est temporairement « emprunté » à l'arbre.

La permission supplémentaire `s`, fournie à `iter`, puis à `f` lors de chaque appel, permet à `f` de faire des effets de bord - si nécessaire - sur un objet mémoire dont la possession est représentée par `s`. Finalement, `f` renvoie un booléen qui permet de stopper l'itération en cours de route. Le booléen renvoyé par `iter` indique si l'itération a été interrompue, ou s'est terminée normalement.

Le code de `iter` est sans surprise, le typeur effectuant automatiquement les manipulations de permissions nécessaires.

```
val rec iter [a, s : perm] (
  f : (a | s) -> bool,
  t : tree a
| s): bool =
```

```
match t with
| Leaf ->
  true
| Node ->
  iter (f, t.left) && f t.elem && iter (f, t.right)
end
```

3.1.3 Inversion de contrôle

Lors de l'itération en utilisant une fonction d'ordre supérieur, c'est celle-ci (le producteur) qui appelle le code utilisateur (le consommateur (`f`)). Dans le cas d'un itérateur, c'est l'inverse : c'est le consommateur qui invoque le producteur le code d'itération quand il a besoin d'un nouvel élément, via la fonction `next`.

La littérature décrit des façons d'effectuer la conversion de l'un vers l'autre de manière semi-automatique, fournissant ainsi, à partir d'une fonction d'itération, l'implémentation de l'itérateur correspondant. La transformation en question, expliquée dans [6], correspond à une mise en forme CPS [13] suivie d'une défonctionnalisation [14]. La technique de défonctionnalisation est toutefois décrite pour un système de type "classique" (System-F et GADTs), différent de celui de Mezzo, et ne s'applique donc pas directement ici.

Ma démarche a été finalement de me restreindre à un itérateur sur le type arbre tel que défini listing 3, et de déterminer à la main l'algorithme pour l'itérateur, en s'inspirant des résultats donnés par les méthodes automatiques précédemment évoquées.

3.2 Itérateurs comme types de données abstraits

L'écriture d'un itérateur (sur arbre) en Mezzo est difficile pour deux raisons : l'implémentation, et l'interface.

Comme évoqué précédemment, l'*implémentation*, un peu plus subtile que pour la fonction `iter`, est difficile à **typer** en Mezzo. Au lieu de se reposer sur la pile d'exécution, il est nécessaire d'exhiber une structure mémorisant la progression de l'itération, entre deux appels à `next`, et d'encoder cette propriété avec des permissions.

L'écriture de l'*interface* est également non triviale. Tout d'abord, pour pouvoir itérer sur l'arbre, l'itérateur doit posséder la permission d'accéder à celui-ci et à ses éléments. La permission `t @ tree a` est donc *consommée* lors de la création de l'itérateur, et **doit** être *restaurée* quand l'itération est finie.

Par ailleurs, retourner un élément d'un nœud de l'arbre implique de retourner la permission associée, afin de permettre à l'utilisateur d'y accéder. Après avoir retourné cette permission, l'itérateur est « *troué* » : il ne possède plus l'arbre entier. Il est donc nécessaire que l'utilisateur redonne la permission sur l'élément avant de continuer l'itération.

3.2.1 Implémentation : Algorithme

La structure interne choisie consiste en une liste des sous-arbres restant à explorer. Celle-ci, initialement, ne contient que `t`, où `t` est l'arbre à parcourir. À chaque appel à `next`, l'élément en tête de liste est examiné : s'il s'agit d'un nœud, l'élément qu'il

contient est retourné, et les fils gauche et droit sont ajoutés à la liste à la place du nœud. S'il s'agit d'une feuille, on l'enlève simplement de la liste et on recommence.

Ceci correspond à un parcours infixe, mais est facilement transposable en un parcours préfixe ou suffixe.

3.2.2 Interface

On écrit l'itérateur dans le style « type abstrait et fonctions agissant dessus ». Notre type itérateur est paramétré par le type de ses éléments, et par une permission `post`, correspondant à la permission sur la collection. Les signatures des fonctions créant un nouvel itérateur et stoppant l'itération sont sans surprises.

```
abstract tree_iterator a (post : perm)

val new : [a]
  (consumes t : tree a) ->
  tree_iterator a (t @ tree a)

val stop : [a, post : perm]
  (consumes it : tree_iterator a post) -> (| post)
```

Une première contrainte est qu'il faut effectivement pouvoir retourner la permission `post` à la fin de l'itération, puisqu'elle correspond à la possession de l'arbre au complet. Or, la structure de données interne ne contient que la partie de l'arbre *qui reste à explorer*. La partie *déjà explorée* n'est pas stockée, car l'implémentation n'en a pas besoin [3].

Il nous faudrait donc pouvoir écrire la permission décrivant un « segment d'arbre ». C'est l'approche utilisée dans [2], dans le cas d'un itérateur sur liste, en logique de séparation. Cependant, cette méthode ne peut pas être utilisée directement en Mezzo, car il n'est pas possible de définir des types somme non taggée.

On utilise alors un autre encodage, qui caractérise cette permission comme étant la permission qui, ajoutée à celle sur l'état interne, permet d'obtenir `post`. Cette propriété a la forme d'une implication, nous utilisons donc une fonction qui n'a pas d'effet à l'exécution, et dont le rôle et d'exprimer cette transformation de permissions. Cette fonction mime en fait la *magic-wand* de la logique de séparation [2].

La deuxième contrainte, indiquant qu'il est nécessaire de redonner la permission sur l'élément « prêté » à l'utilisateur, est satisfaite selon le même schéma : une fonction sera renvoyée, servant à réintégrer la permission sur l'élément en un itérateur complet.

On peut alors écrire la signature de la fonction `next`, demandant un nouvel élément à l'itérateur.

```
val next [a, post : perm]
  (consumes it : tree_iterator a post) ->
  either (focused a (it @ tree_iterator a post))
         (| post)
```

3. Même en stockant les nœuds déjà parcourus (ce qui nous donne un zipper, finalement), il n'est pas plus facile de convaincre le typeur que la réunion des permissions sur les deux morceaux redonne la permission pour l'arbre entier.

L'appel "`next it`" nécessite la permission `(it @ tree_iterator a post)` et… la consomme. Immédiatement après l'appel à `next`, l'itérateur n'est plus directement utilisable : on lui a emprunté un élément.

À la place, `next` retourne :

- Soit un élément `x` de type `a`, associé à une fonction permettant d'obtenir à nouveau `(it @ tree_iterator a post)` à condition de redonner `(x @ a)`.
- Soit la permission `post`, car l'itérateur ne contient plus d'éléments et s'est arrêté.

```
data either a b =
| Left  { contents : a }
| Right { contents : b }
```

Une valeur de type "`focused a post`" correspond ainsi au couple d'une valeur `x` de type `a`, et d'une « baguette magique » permettant de convertir `(x @ a)` en `post`.

```
 alias focused a (post : perm) =
  (x : a, release : wand (x @ a) post)
```

Comme annoncé, la notion de « baguette magique » est simulée par une fonction Mezzo qui n'a aucun effet à l'exécution. Ainsi, "`wand pre post`" consomme `pre` et produit `post`. Elle est en outre à usage unique : on exprime ce fait en munissant la « baguette » d'une « munition » que l'on assure unique en la quantifiant existentiellement, grâce au même procédé qu'exposé en section 2.2.

```
alias wand (pre : perm) (post : perm) =
  {ammo : perm} (
    (| consumes (pre * ammo)) -> (| post)
  | ammo)
```

Listing 5 : Définition de `wand`

Cette munition aura également une utilité réelle : elle permet d'abstraire une permission nécessaire pour obtenir `post`, mais que l'on ne veut pas avoir à citer dans `pre` [4] (par exemple, parce qu'on ne sait pas l'exprimer en tant que telle !).

3.2.3 Implémentation : Mezzo

Le type de l'itérateur lui-même consiste en la structure interne, et la promesse qu'il est possible de convertir la permission sur cette structure en `post`.

```
alias tree_iterator a (post : perm) =
  ref (focused (list (tree a)) post)
```

L'idiome `focused` est utilisé encore une fois ici. Ce qui joue le rôle de la « munition », permission abstraite contenue dans l'itérateur et nécessaire pour obtenir `post`, est la permission sur les éléments déjà consommés. Celle-ci est bien caractérisée par le

4. Dans une précédente version des itérateurs, certaines permissions n'étaient pas quantifiées existentiellement : elles paramétraient le type de l'itérateur. Cela posait des difficultés pour parler d'un type "itérateur" en général, le type étant paramétré par des permissions dépendant de son état interne.

fait qu'en l'ajoutant à celle sur la structure, on obtient `post`, et l'on a pas besoin de savoir l'expliciter tout au long du programme, puisqu'elle est abstraite par une quantification existentielle.

```
val new [a] (consumes t : tree a): tree_iterator a (t @ tree a) =
  newref (
    Cons { head = t; tail = Nil },
    fun ()  : () = ()
  )
```

Initialement, la liste contient directement `t` : la permission sur la liste se convertit automatiquement en permission sur `t`. `wand` sera donc empaquetée avec le témoin `empty` en guise de `ammo`.

```
val stop [a, post : perm]
  (consumes it : tree_iterator a post):
  (| post) =
  let _, release = !it in
  release ()
```

Stopper l'itération est également aisé : il suffit d'appeler la fonction « baguette magique » : celle-ci nous fournit `post`.

C'est dans `next` que les subtilités interviennent. Le code complet de la fonction est en annexe A.3 : étudions uniquement le cas où la liste interne (`Cons { head = t; tail = ts }`) contient un arbre non vide en tête (`Node { left; elem; right }`).

Une petite précision avant de commencer : cette partie est plutôt technique. Elle met en jeu un nombre élevé de permissions sur lesquelles, en pratique, le typeur Mezzo fait également beaucoup de transformations (découpage/regroupement). Cela rend l'analyse du code peu simple à comprendre sur papier ; cependant, il n'est pas nécessaire de comprendre tous les détails.

La liste interne est d'abord modifiée pour remplacer l'arbre de tête par ses fils. Il s'agit ensuite de définir une fonction `surrender` de type `wand`, permettant de convertir `(elem @ a)` en `(it @ tree_iterator a post)` lorsque l'utilisateur « rend » l'élément. Celle-ci prend en argument la permission `(elem @ a)`, et une certaine permission `ammo` : dans notre cas, il s'agit en fait des permissions sur les morceaux restants de l'itérateur, nécessaires pour le reconstituer.

> **Point Mezzo 10** *Si la gestion automatique des quantifications existentielles fonctionne la plupart du temps comme on le souhaite, les codes manipulés ici commencent à pousser le typeur dans des retranchements où il lui arrive de faire des mauvais choix*[5].
>
> *Mezzo fournit pour cela des mécanismes permettant une gestion manuelle.*
>
> - `let flex x: κ in …` *: introduit une nouvelle variable* `x` *dite flexible, dont l'utilité est de pouvoir s'unifier avec le témoin abstrait d'une existentielle dépaquetée précédemment.*

> - `pack {x:κ} y witness x'` *: permet d'empaqueter un type quantifié existentiellement en fournissant manuellement un témoin (*`x'`*). La construction* `pack x @ t witness u` *est également reconnue, dans le cas où* `t` *et un type quantifié existentiellement.*
>
> *Ces fonctionnalités ont en fait été ajoutées à Mezzo pendant mon stage afin de pouvoir faire typer mes expérimentations sur les itérateurs.*

```
let flex p : perm in
let surrender (|
  consumes (
    elem @ a *
    left @ tree a * right @ tree a *
    t @ Node { left : =left; elem : =elem; right : =right } *
    ts @ list (tree a) *
    it @ Ref { contents : (=stack, =release) } *
    p
  )
): (| it @ tree_iterator a post) =
  ...
```

La variable flexible `p` sera ici unifiée avec le témoin `ammo` de la baguette magique de l'itérateur (nommée `release` dans le code). Or celle-ci correspond à la permission sur les *éléments déjà consommés*.
Ainsi, on passe en argument à `surrender` les permissions :

- `(elem @ a)` : l'élément courant
- `p` : les permissions pour les éléments déja consommés
- `ts @ list (tree a)` : la queue de la liste
- Les permissions sur les morceaux de structure ayant été examinés à cette étape.

On se convainc qu'en « assemblant » ces morceaux (ce que Mezzo sait faire), il est possible de reconstituer la permission `it @ tree_iterator a post`.

L'itérateur ayant avancé d'un cran, il faut ajouter la permission `elem @ a` à la grosse permission abstraite correspondant aux éléments consommés, et mettre à jour le type de la baguette magique. C'est ce que fait `surrender`, en empaquetant à la main le type de la baguette magique `release` avec un nouveau témoin, contenant `p` et les éléments consommés, dont `elem @ a`.

```
  pack release @ wand (stack @ list (tree a)) post
  witness
    elem @ a *
    t @ Node { left; elem; right } *
    p
```

5. Considérons le type `{p: perm} (| p)` par exemple : si une permission `q` est un témoin satisfaisant pour cette quantification existentielle, alors `q * r` l'est aussi. Le problème est qu'après « empaquetage », les permissions témoins sont abstraites et ont été enlevées de l'environnement : si `q * r` a été choisi comme témoin, `r` n'est plus accessible alors qu'il aurait pu être nécessaire ultérieurement.

Notez que la fonction `release` est toujours la même : il s'agit de la fonction `fun () : () = ()` crée par `new`. Cependant, son **type** change. Mezzo sait reconnaître que la conversion est légale, puisqu'il s'agit uniquement de transférer des permissions de `pre` vers `ammo` (cf. la définition de `wand`, listing 5).

Bilan Nous avions deux problèmes : rendre compte d'une permission non définissable telle quelle en Mezzo, et exprimer dans les types une logique « si l'on me donne telle permission, alors je peux continuer d'itérer ».

Ceux-ci sont résolus en utilisant conjointement deux « idiomes » Mezzo : tout d'abord la possibilité d'abstraire une permission en la quantifiant existentiellement permet de stocker une permission sans la définir, en la construisant itérativement. Ensuite, l'utilisation de fonctions « baguette magique », permettant d'exprimer des propriétés non triviales de conversion de permissions.

Le résultat obtenu présente des aspects positifs comme négatifs. Tout d'abord, il est satisfaisant de constater que l'on a bien pu typer en Mezzo l'implémentation qui était envisagée au premier abord. Par ailleurs, nos itérateurs présentent une interface relativement simple : les types vus par l'utilisateur lors de leur utilisation ne laissent pas présager de la complexité interne. On peut donc espérer qu'utiliser de tels itérateurs sera relativement aisé. Cependant, implémenter l'itérateur lui-même demande beaucoup d'annotations manuelles, qui nécessitent une (très) bonne compréhension du fonctionnement du typeur Mezzo pour être lues, voire écrites.

3.3 Itérateurs généralistes et encodage objet

Nous avons écrit un itérateur pour un type de données particulier. La méthode peut s'appliquer pour d'autres structures, mais il serait pénible de devoir redéfinir à chaque fois des opérations agissant sur l'itérateur. On veut donc pouvoir écrire du code agissant sur « un itérateur », indépendamment de son implémentation.

Il n'y a pas de mécanisme tel que les foncteurs en Mezzo. Toutefois, nous avons déjà vu un mécanisme permettant d'abstraire une implémentation en Mezzo : la quantification existentielle.

Une solution est tout d'abord d'adopter un style *Orienté-Objet* (OO) : un itérateur est un objet équipé de deux méthodes, `next` et `stop`. Ces méthodes ont besoin d'accéder à l'état interne de l'itérateur. On regroupe alors dans un type de données les deux méthodes, et l'état interne `s`, sous la forme d'une permission.

```
data iterator_s (s : perm) a (post : perm) =
  Iterator {
    next : (| consumes s) -> either (focused a s)
                                    (| post);
    stop : (| consumes s) -> (| post)
| s }
```

Dans le cas de notre itérateur sur arbre, la permission `s` correspond à `(it @ iterator a post)` pour un certain terme `it`.

On abstrait enfin sur l'implémentation particulière, et donc sur l'état interne.

```
alias iterator a (post : perm) =
  {s : perm} iterator_s s a post
```

Cet encodage orienté-objet est dans le style de [3], à la différence près que l'état interne est une permission et non un type. Dès l'arrivée d'un type `(iterator a post)` dans l'environnement, la quantification existentielle est dépaquetée, la permission `s` libérée dans l'environnement, et il est possible d'appeler directement `it.next()` ou `it.stop()`. Cet encodage s'exprime ainsi plutôt agréablement en Mezzo.

Par ailleurs, cette signature exprime bien le protocole objet d'un itérateur (figure 4) : l'appel à `next` ou `stop` consomme l'état interne. Il est par exemple impossible d'appeler `it.stop()` deux fois à la suite.

Conversions Après avoir implémenté un itérateur en style « type de données abstrait » (comme `tree_iterator`), il est possible de le convertir en un itérateur « généraliste », style OO. Cela est réalisé par la fonction `wrap` [6], prenant en argument un terme de type `i` quelconque, associé à deux fonctions `next` et `stop`.

```
val wrap [a, i, post : perm] (
  consumes it : i,
  next : (consumes it : i) -> either (focused a (it @ i)) (| post),
  stop : (consumes it : i) -> (| post)
) -> iterator a post
```

La conversion inverse est également possible, en appelant simplement les méthodes de l'itérateur.

Opérations sur itérateurs généralistes L'intérêt de tels itérateurs généralistes, que l'on peut obtenir à partir d'une implémentation quelconque, est que d'éventuelles fonctions de manipulation sur ces itérateurs seront également génériques.

Un itérateur se comportant comme un flux, les opérations standard sur ceux-ci peuvent s'implémenter sur les itérateurs. Par exemple, on peut écrire une fonction `filter`, qui, à partir d'un itérateur, construit un nouvel itérateur retournant uniquement les éléments satisfaisant un certain prédicat.

```
val filter [a, p : perm, post : perm] (
  consumes it : iterator a post,
  f : (a | p) -> bool
| consumes p) -> iterator a (p * post)
```

Cette fonction consomme la permission sur l'itérateur d'origine : celle-ci fait alors partie de l'état interne du nouvel itérateur.

Par le même procédé que pour la fonction `iter` d'ordre supérieur (listing 4), la fonction `f` fournie par l'utilisateur peut avoir un état interne. Elle y accède via la permission `p`, fournie lors de la création de l'itérateur, et redonnée lors de son arrêt en même temps que `post`.

L'implémentation de `filter` est fournie en annexe A.4 : elle utilise également des mécanismes permettant d'« aider » le typeur (`let flex`, `assert`) mais dans une moindre

6. Le code complet de `wrap` est disponible en annexe A.4

mesure, puisqu'il n'y a pas d'empaquetage d'existentielles à faire manuellement. D'autres opérations sur itérateurs ont également été implémentées, incluant `map`, `zip`, `concat`, `equal`, ...

Ces implémentations d'itérateurs sur listes et arbres, ainsi que des fonctions de manipulation génériques sont disponibles dans les sources de Mezzo [9] (`stdlib/list.mz`, `stdlib/mutableTreeMap.mz`, `stdlib/iterator.mz`).

3.4 Limitations

Multiples *wand*s L'implémentation courante des itérateurs utilise de multiples fonctions « baguette magique » : celle contenue dans l'itérateur reste la même, mais chaque élément renvoyé par `next` est accompagné d'une fonction *wand* différente. Ceci nécessite d'allouer en mémoire une cellule pour représenter le couple *(élément, wand)* au lieu de renvoyer seulement l'élément. Cette contrainte, qui a un impact sur les performances, n'est pas due à l'implémentation, puisque les fonctions *wand* n'ont d'effet que sur les permissions, qui sont effacées à l'exécution. On aimerait donc pouvoir s'en affranchir.

Une idée, qui a effectivement été essayée, est de n'avoir qu'une seule fonction `w`, d'implémentation `fun (): () = ()`, et de remplacer chaque création d'une fonction de type `wand pre post` par le renvoi d'une permission `w @ wand pre post`. On se retrouve ainsi avec de nombreux types `wand x y` potentiellement très différents sur la même fonction `w`.
Cela demande au typeur de deviner le type qui convient à chaque utilisation de `w`, ce qui rend le typage assez lourd. Par ailleurs, cette méthode nécessite que toutes les conversions de permissions que l'on voudra exprimer via la wand puissent être faites par le typeur Mezzo sans aide : on ne peut pas redéfinir l'implémentation de `w` pour inclure des empaquetages/dépaquetages manuels d'existentielles par exemple. En pratique, on ne peut écrire la fonction `zip` par exemple.

Expressivité En l'état, nos itérateurs sont plus restreints qu'il n'est en réalité nécessaire. On peut imaginer des situations où l'utilisateur voudrait itérer de manière un peu compliquée sur sa structure, en gardant par exemple plusieurs permissions sur des éléments, puis en les rendant en bloc à la fin. Ceci n'est pas autorisé actuellement, puisque l'utilisateur ne peut détenir qu'un seul élément à la fois.

On aimerait par ailleurs ajouter une fonction `hasNext` pour plus de confort : en l'état, on ne sait pas écrire de fonction `last` qui renverrait le dernier élément de l'itérateur ! On veut également pouvoir typer le fait qu'après un `hasNext`, si la réponse est `false`, il est illégal d'appeler `next`. C'est une piste que je n'ai pas encore totalement explorée [7].

3.5 Développements

D'autres techniques existent pour passer le contrôle, implémentées dans certains langages de programmation afin de permettre aux utilisateurs d'écrire simplement des itérateurs. On peut penser à la notion de coroutine [4].

7. Il y a une subtilité : on veut pouvoir passer de manière transparente d'un état « j'ai appelé `hasNext` et je sais si l'itérateur est vide » à l'état « je n'ai pas appelé `hasNext` »... (ce qui reviendrait à avoir une sorte d'ε-transition dans l'automate représentant le protocole objet)

Une autre technique consiste à utiliser des threads, et à exécuter le consommateur (qui lit les éléments) et le producteur (qui a accès à la structure) dans deux threads séparés, et de les faire communiquer par des canaux. L'intérêt est que l'on peut écrire à la fois le producteur et le consommateur dans un style direct (pas besoin de structure interne simulant la pile d'exécution comme pour nos itérateurs), le passage de contrôle se faisant grâce aux primitives de synchronisation sur les canaux.

Le protocole de communication serait alors le suivant : le producteur envoie chaque élément `x` de type `a` dans un canal. Le consommateur répond, une fois qu'il a fini, en renvoyant un message de type `(| x @ a)` (message *ack*). La difficulté est que les types des messages *ack* sont tous différents, et dépendent de l'élément envoyé précedemment. On ne peut donc simplement créer un canal par lequel tous les *ack* transiteraient.

Plusieurs tentatives ont été faites dans ce sens : François Pottier a écrit en Mezzo trois versions implémentant ce protocole. La première utilise deux canaux (interface en annexe A.5.2) et nécessite que la permission `(x @ a)` soit duplicable, la deuxième crée un canal pour chaque *ack*, et la troisième, particulièrement intéressante, utilise la structure (axiomatisée) de *tube* (annexe A.5.3) pour échanger les *ack* à côté du canal principal. Un *tube* est une sorte de canal dans lequel il est possible d'envoyer des messages de type hétérogène. Le code est en annexe A.5.1.

J'ai écrit pour ma part un prototype qui axiomatise un canal permettant l'échange bidirectionnel des éléments et des messages *ack*. Il utilise de manière intensive la possibilité d'avoir plusieurs « types flèche » (types de la forme `a -> b`) sur le même terme, et s'inspire de [5]. Son interface, ainsi qu'un exemple de code l'utilisant, implémentant un couple producteur/consommateur pour l'itération sur un arbre, sont disponibles en annexe A.6. On peut noter que des deux côtés, producteur et consommateur, le code est direct et sans complications.

Cependant, il est pour l'instant fort peu clair de la manière dont il est souhaitable d'intégrer la notion de canaux au langage (et à la preuve de correction en Coq).

4 Perspectives futures

L'utilisation de Mezzo au long du stage, notamment lors de l'écriture des itérateurs, a donné des idées de fonctionnalités qu'il pourrait être utile d'intégrer dans le langage.

Canaux Des canaux ajoutés en primitive au langage seraient peut-être utiles ; leur nature exacte est pour l'instant incertaine. Actuellement, des canaux peuvent être implémentés en Mezzo en utilisant le mécanisme de verrous qui, lui, est intégré au langage. Cependant, un canal implémenté de cette manière ne peut faire transiter que des éléments du même type : on ne peut s'en servir pour implémenter le protocole décrit dans la section 3.5, puisqu'il nécessite des types de retours (`(| x @ a)`) variant selon la valeur envoyée précédemment.

Code *ghost* Une idée permettant de résoudre élégamment le problème des multiples *wand*s est d'intégrer dans le typeur la notion de fonction « fantôme » : une fonction « fantôme » serait une fonction manipulant uniquement des permissions et n'ayant aucun effet à l'exécution. Il serait alors possible d'effacer ces fonctions à l'exécution ou à la compilation, en même temps que les permissions. Les fonctions *wand* correspondant à cette description, en avoir une multitude ne serait alors plus un problème.

Cela simplifierait même potentiellement le code des algorithmes écrits en Mezzo, puisque actuellement il faut faire attention, lors de la manipulation de *wand*s, de ne pas en allouer trop souvent, afin de ne pas changer la complexité des algorithmes (d'où la situation où on change le type d'une même fonction *wand*). Avec des fonctions *ghost*, le problème ne se pose plus puisqu'elles disparaîtront à l'exécution.

Conclusion

Ce stage m'a permis de mieux comprendre les enjeux d'un langage disposant d'un système de types très expressif, Mezzo, plus particulièrement.

La problématique du stage consistait à comprendre comment exprimer, dans son système de types, des protocoles objet suivis par des implémentations connues.
Le jeu en vaut la chandelle puisque alors le respect du protocole est garanti statiquement. Un non-respect de celui-ci sera rejeté à la compilation, ce qui permet d'éliminer une classe d'erreurs de programmation souvent subtiles, et que l'on a pas l'habitude de voir détectées grâce au langage de programmation.

Contributions J'ai abordé la question sous deux approches différentes : tout d'abord dans le cas d'une implémentation externe, où seule l'interface est écrite en Mezzo. Pour ce cas, j'ai écrit une interface Mezzo pour les API Posix fichiers et sockets, et en ai extrait une méthode qui se généralise à une certaine classe de protocoles.

La deuxième approche, pour laquelle l'implémentation est également faite en Mezzo, est envisagée par l'exemple, avec les itérateurs. J'ai ainsi implémenté des itérateurs sur listes et arbres (que l'on retrouve dans `stdlib/list.mz`, `stdlib/mutableTreeMap.mz`), remplaçant de précédents itérateurs très difficiles à utiliser (dans le cas des listes) ou non *type-safe* (dans le cas des arbres).
Par ailleurs, j'ai écrit une petite bibliothèque permettant de manipuler ces itérateurs de manière générique.

S'il s'agit bien d'un exemple particulier, on peut en extraire de celui-ci des "design patterns", techniques de programmation applicables pour résoudre un problème particulier - notamment l'utilisation de "magic wands" pour exprimer, avec des permissions, des propriétés non triviales de l'algorithme implémenté. J'espère ainsi avoir éclairé un peu la manière dont il est possible, en Mezzo, d'exprimer et d'implémenter des protocoles objet.

Une dernière remarque que l'on peut faire est que, pour l'instant, implémenter des algorithmes en Mezzo peut être difficile. La nécessité de devoir parfois « aider » le typeur en fournissant des annotations manuelles (par exemple, lors de l'utilisation de quantifications existentielles) rend obligatoire une compréhension fine du fonctionnement du typeur. L'expressivité de Mezzo, en plus des avantages qu'elle fournit, a un coût certain.

Articles étudiés

[1] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas Gross, *Statically Checking API Protocol Conformance with Mined Multi-Object Specifications*, 2012.

[2] Neelakantan R. Krishnaswami, Jonathan Aldrich, Lars Birkedal, Kasper Svendsen, and Alexandre Buisse, *Design patterns in separation logic*, Types in Language Design and Implementation (2009), 105–116.

[3] Benjamin C. Pierce and David N. Turner, *Simple Type-Theoretic Foundations for Object-Oriented Programming*, 1993.

[4] Ana Lucia de Moura and Roberto Ierusalimschy, *Revisiting Coroutines*, ACM Trans. Program. Lang. Syst. (2009), 6 :1–6 :31.

[5] Jules Villard, Étienne Lozes, and Cristiano Calgagno, *Proving Copyless Message Passing*, Asian Symposium on Programming Languages and Systems (2009).

[6] Gabriel Scherer, *Generators, iterators, control and continuations* : `http://gallium.inria.fr/blog/generators-iterators-control-and-continuations/`, 2013.

[7] *POSIX Manpages*.

[8] *Scala Documentation : Iterator* : `http://www.scala-lang.org/api/current/index.html#scala.collection.Iterator`.

[9] *Sources de Mezzo* : `http://gallium.inria.fr/~protzenk/mezzo-lang/proto/mezzo-latest.tar.bz2`.

Articles de référence

[10] François Pottier and Jonathan Protzenko, *Programming with permissions in Mezzo*, ICFP (Septembre 2013).

[11] John Reynolds, *Separation Logic : A Logic for Shared Mutable Data Structures*, Logic In Computer Science (2002), 55–74.

[12] Nels E. Beckman, Duri Kim, and Jonathan Aldrich, *An Empirical Study of Object Protocols in the Wild*, ECOOP (2011), 2–26.

[13] Olivier Danvy, *Three Steps for the CPS Transformation*, Departement of Computing and Information Sciences, Kansas State University, 1991.

[14] François Pottier and Nadji Gauthier, *Polymorphic Typed Defunctionalization*, Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04) (2004), 89–98.

A Annexe

A.1 connected-socket.mzi

```
open inet
open error
open string
open list

data socket_domain =
| PF_UNIX
| PF_INET
| PF_INET6

data sockaddr =
| ADDR_UNIX { name : string }
| ADDR_INET { addr : inet_addr ; port : int }

data socket_type =
| SOCK_STREAM
| SOCK_SEQPACKET

data msg_flag =
| MSG_OOB
| MSG_DONTROUTE
| MSG_PEEK

val domain_of_sockaddr : sockaddr -> socket_domain

abstract can_connect (s : term) (stamp : type): perm
abstract can_bind (s : term) (stamp : type): perm
abstract can_close (s : term) (stamp : type): perm
abstract can_listen (s : term) (stamp : type): perm
abstract can_accept (s : term) (stamp : type): perm
abstract can_send (s : term) (stamp : type): perm
abstract can_recv (s : term) (stamp : type): perm

abstract socket (stamp : type)

(* States *)
alias init = { stamp } (
  s : socket stamp
| can_connect s stamp *
  can_bind s stamp *
  can_close s stamp
)
```

```
alias connected = { stamp } (
  s : socket stamp
| can_send s stamp *
  can_recv s stamp *
  can_close s stamp
)

alias bound = { stamp } (
  s : socket stamp
| can_listen s stamp *
  can_close s stamp
)

alias listening = { stamp } (
  s : socket stamp
| can_accept s stamp *
  can_close s stamp
)

alias errno = int

val new : (
  socket_domain,
  socket_type,
  int
) -> result errno (s : init)

val connect : [k] (
  consumes s : socket k,
  addr : sockaddr
| consumes can_connect s k
) ->
  result
    (* connect failed. We stay at the same state *)
    (errno | s @ socket k * can_connect s k)
    (* Success *)
    (| s @ connected)

val bind : [k] (
  consumes s : socket k,
  addr : sockaddr
| consumes can_bind s k
) ->
  result
    (* bind failed. We stay at the same state *)
```

```
    (errno | s @ socket k * can_bind s k)
    (* Success *)
    (| s @ bound)

val listen : [k] (
  consumes s : socket k,
  max_req : int
| consumes can_listen s k
) ->
  result
    (* listen failed. We stay at the same state *)
    (errno | s @ socket k * can_listen s k)
    (* Success *)
    (| s @ listening)

val accept : [k] (
  consumes s : socket k
| consumes can_accept s k
) ->
  result
    (* accept failed *)
    (errno | s @ socket k * can_accept s k)
    (* Success. We return a new socket [con_sock], connected to the client *)
    (con_sock : connected, sockaddr | s @ listening)

val send : [k] (
  s : socket k,
  buf : string,
  ofs : int,
  len : int,
  flags : list msg_flag
| can_send s k
) -> result errno ()

val recv : [k] (
  s : socket k,
  buf : string,
  ofs : int,
  len : int,
  flags : list msg_flag
| can_recv s k
) -> result errno ()

val close : [k] (
  consumes s : socket k
| consumes can_close s k
```

```
) -> result errno ()
```

A.2 idioms.mz

```
(* Idioms. *)

data either a b =
  | Left  { contents : a }
  | Right { contents : b }

alias wand (pre : perm) (post : perm) =
  {ammo : perm} (
    (| consumes (pre * ammo)) -> (| post)
  | ammo)

alias focused a (post : perm) =
  (x : a, release : wand (x @ a) post)
```

A.3 tree-iterator.mz

```
open idioms

(* Trees. *)

data mutable tree a =
  Leaf
| Node { left : tree a; elem : a; right : tree a }

val rec iter [a, p  : perm] (
  f : (    a | p) -> bool,
  t : tree a | p)   : bool =
  match t with
  | Leaf ->
      true
  | Node ->
      iter (f, t.left) && f t.elem && iter (f, t.right)
  end

(* ------------------------------------------------------------------------ *)

(* ADT-style tree iterators *)

alias tree_iterator a (post : perm) =
  ref (focused (list (tree a)) post)
```

```
val new [a] (consumes t : tree a): tree_iterator a (t @ tree a) =
  newref (
    Cons { head = t; tail = Nil },
    fun ()  : () = ()
  )

val rec next [a, post : perm]
  (consumes it : tree_iterator a post):
  either (focused a (it @ tree_iterator a post)) (| post) =

  let stack, release = !it in

  (* [stack] is the list of sub-trees that remain to be explored. *)
  (* assert stack @ list (tree a); *)
  (* [release] is the magic wand that abandons the stack and yields
     the ownership of the entire tree. *)
  (* assert release @ wand (stack @ list (tree a)) post; *)

  match stack with
  | Nil ->
      (* The stack is empty. We are done. The iterator auto-stops. *)
      release ();
      Right { contents = () }

  | Cons { head = t; tail = ts } ->
      (* The stack is non-empty. *)
      match t with
      | Leaf ->
          (* The head tree is empty. Pop it off. *)
          it  := (ts, release);
          (* Persuade the type-checker takes us from the new stack
             to the old stack and (hence) to [post]. *)
          let flex p : perm in
          pack release @ wand (ts @ list (tree a)) post
          witness p * stack @ Cons { head : Leaf; tail = ts };
          (* Continue. *)
          next it

      | Node { left; elem; right } ->
          (* The head tree has a root node. Split this node and push its
             sub-trees onto the stack. *)
          let stack = Cons { head = left; tail = Cons { head = right; tail = ts }} in
          it  := (stack, release);

          (* Now, we must construct a new magic wand, which packs some of our current
```

```
            permissions, and ensures that once the client gives [elem @ a] back to us,
            we can produce [it @ tree_iterator a post]. *)
          let flex p : perm in
          let surrender (|
            consumes (
              left @ tree a *
              elem @ a *
              right @ tree a *
              t @ Node { left; elem; right } *
              ts @ list (tree a) *
              it @ Ref { contents : (=stack, =release) } *
              p
            )
          ): (| it @ tree_iterator a post) =

            (* We know that the new [stack] is [left :: right :: ts], because this is
               a duplicable fact. Similarly, we know that the old [stack] was [t :: ts].
               Thus, if we own the new [stack] as a list of trees, and if own the node [t]
               and its element, then we can reconstruct that we own the old [stack] as a
               list of trees, and from there, apply [release] -- at its previous type --
               in order to obtain [post]. This justifies the following new view of
               [release]. *)

            pack release @ wand (stack @ list (tree a)) post
            witness
              elem @ a *
              t @ Node { left; elem; right } *
              p;

            (* This assertion is superfluous. It says that, once the client surrenders
               [elem @ a], we again have a well-formed new [stack] and that [release]
               allows us to abandon this stack and obtain [post]. *)
            assert
              stack @ list (tree a) *
              release @ wand (stack @ list (tree a)) post;
            ()
          in

          (* This assertion is superfluous. *)
          assert surrender @ wand (elem @ a) (it @ tree_iterator a post);

          Left { contents = (elem, surrender) }
      end
    end

val stop [a, post : perm]
```

```
  (consumes it : tree_iterator a post):
  (| post) =
  let _, release = !it in
  release()
```

A.4 oo-iterator.mz

```
open idioms

(* Generic OO-style iterators *)

data iterator_s (s : perm) a (post : perm) =
  Iterator {
    next : (| consumes s) -> either (focused a s)
                                    (| post);
    stop : (| consumes s) -> (| post)
| s }

alias iterator a (post : perm) =
  {s : perm} iterator_s s a post

(* Wrapping an ADT-style iterator into a generic (OO-style) iterator. *)

val wrap [a, i, post : perm] (
  consumes it : i,
  next : (consumes it : i) -> either (focused a (it @ i)) (| post),
  stop : (consumes it : i) -> (| post)
): iterator a post =

  (* A convenient abbreviation. *)
  let alias s : perm = it @ i in

  (* Perform the application of the ADT functions to the iterator [i],
     so as to obtain an object whose internal state is inaccessible. *)
  Iterator {
    next = (fun (| consumes s): either (focused a s) (| post) = next it);
    stop = (fun (| consumes s): (| post) = stop it)
  }

(* A [filter] function on generic iterators. *)

val filter [a, p : perm, post : perm] (
  consumes it : iterator a post,
  f : (a | p) -> bool
| consumes p)  : iterator a (p * post)
```

```
=

  (* A name for the (unpacked) internal state of the underlying iterator. *)
  let flex underlying : perm in
  assert it @ iterator_s underlying a post;

  (* An abbreviation for the internal state of the new iterator.
     It contains the underlying iterator and the permission [p],
     which represents the internal state of [f]. The underlying
     iterator is in an unpacked state. *)
  let alias s  : perm = p * underlying in

  let rec next (| consumes s)  : either (focused a s) (| post * p) =
    let e = it.next() in
    match e with
    | Right ->
        (* The underlying iterator is finished. We are finished as well. *)
        e
    | Left { contents = (x, surrender) } ->
        (* The underlying iterator produces [x], together with a magic
           wand [surrender], which we must use to signal that we are done
           with [x]. *)
        if f x then begin
          (* We would like to keep [x]. *)
          assert p * surrender @ wand (p * x @ a) (p * underlying);
          (* The second assertion is superfluous, but explains what is going on. *)
          assert surrender @ wand (x @ a) (p * underlying);
          e
        end
        else begin
          (* We would like to skip [x]. Signal that we are done with it,
             and look for the next element. *)
          surrender();
          next()
        end
    end
  in

  Iterator { next; stop = it.stop }
```

A.5 tree-coroutine.mz, channel.mzi, tube.mzi

Le code de cette section a été écrit par François Pottier.

A.5.1 tree-coroutine.mz

```
data mutable tree a =
  Leaf
| Node { left : tree a; elem : a; right : tree a }

open channel

(* ------------------------------------------------------------------------- *)

(* This function iterates over the tree, and submits the elements onto the
   channel [pipe]. It then waits for an acknowledgement to arrive on the
   channel [ack], and proceeds with the iteration. *)

(* The code is too simplistic, insofar as there is no way for the producer
   to signal that the iteration is finished. Maybe we should allow a channel
   to be closed, in which case further calls to [send] or [receive] would
   fail at runtime? *)

(* Unfortunately, this approach is limited to the case where [a] is duplicable,
   because we cannot express the idea that [ack] transmits [x @ a] where [x] is
   the last element that was sent via [pipe]. *)

val rec iter [a] duplicable a => (t : tree a, pipe : channel a, ack : channel bool)  : bool =
  match t with
  | Leaf ->
      true
  | Node ->
      iter (t.left, pipe, ack) &&
      begin send (pipe, t.elem); receive ack end &&
      iter (t.right, pipe, ack)
  end

(* Here is a client that receives the elements and counts them. Note that it
   has local state. *)

val count [a] (pipe : channel a, ack : channel bool)  : () =
  let c = newref 0 in
  preserving c @ ref int while true do begin
    let x = receive pipe in
    incr c;
    send (ack, true)
  end
  (* Stupid. This point in the code is unreachable. *)

(* Now, combine the producer and consumer. We cheat by running the producer
```

```
   on the current thread. If we spawned a new thread, we would have to
   transmit [t @ tree a] to it, and get it back afterwards, using [join]. *)

val test [a] duplicable a => (t : tree a)  : bool =
  let pipe, ack = new(), new() in
  thread ::spawn (fun ()  : () =
    count (pipe, ack)
  );
  iter [a] (t, pipe, ack)
    (* Interesting example of a type application that is required. *)

(* ------------------------------------------------------------------------- *)

(* Another approach, where we use a new [ack] channel at every interaction.
   This is a bit wasteful but allows us to typecheck the code even when [a]
   is not duplicable. *)

alias ack_t (p  : perm) =
  channel (bool | p)

alias packet a =
  (x : a, ack_t (x @ a))

alias pipe_t a =
  channel (packet a)

(* TEMPORARY a lot of unpleasant type applications are required in the
   code that follows. *)

val send_and_wait [a] (pipe : pipe_t a, x : a)  : bool =
  let ack  : ack_t (x @ a) = new [(bool | x @ a)] () in
  send [packet a] (pipe, (x, ack));
  receive [(bool | x @ a)] ack

val rec iter [a] (t : tree a, pipe : pipe_t a)  : bool =
  match t with
  | Leaf ->
      true
  | Node ->
      iter (t.left, pipe) &&
      send_and_wait (pipe, t.elem) &&
      iter (t.right, pipe)
  end

(* Here is a client that receives the elements and counts them. Note that it
   has local state. *)
```

```
val count [a] (pipe : pipe_t a)  : () =
  let c = newref 0 in
  preserving c @ ref int while true do begin
    let x, ack = receive [packet a] pipe in
    incr c;
    send [(bool | x @ a)] (ack, true)
  end
  (* Stupid. This point in the code is unreachable. *)

(* Now, combine the producer and consumer. We cheat by running the producer
   on the current thread. If we spawned a new thread, we would have to
   transmit [t @ tree a] to it, and get it back afterwards, using [join]. *)

val test [a] (t : tree a)  : bool =
  let pipe = new() in
  thread ::spawn (fun ()  : () =
    count [a] pipe
      (* Interesting example of a type application that is required. *)
  );
  iter (t, pipe)

(* ------------------------------------------------------------------------- *)

(* A third approach, where [ack] is a tube and is re-used at every interaction. *)

open tube

alias ack_inlet (p  : perm) =
  inlet (bool | p)

alias ack_outlet (p  : perm) =
  outlet (bool | p)

alias packet a (ack  : term) =
  (x : a | ack @ ack_outlet (x @ a))

alias pipe_t a (ack  : term) =
  channel (packet a ack)

(* In this variant of [send_and_wait], we do not create the [ack]
   tube, which already exists. We set it up for one exchange, send
   the outlet permission via the pipe, and keep the inlet permission,
   which is used to receive. After the interaction, the tube is
   inert again. *)
```

```
val send_and_wait [a] (pipe : pipe_t a ack, ack : inert, x : a)  : bool =
  (* Set up the tube for this one interaction. *)
  setup [(bool | x @ a)] ack;
  (* Send [x], together with the outlet permission for [ack], via the pipe. *)
  channel ::send [packet a ack] (pipe, x);
  (* Recover the permission [x @ a] via the inlet side of [ack]. *)
  tube ::receive [(bool | x @ a)] ack
  (* [ack] is now inert again. *)

val rec iter [a] (t : tree a, pipe : pipe_t a ack, ack : inert)  : bool =
  match t with
  | Leaf ->
      true
  | Node ->
      iter (t.left, pipe, ack) &&
      send_and_wait (pipe, ack, t.elem) &&
      iter (t.right, pipe, ack)
  end

(* Here is a client that receives the elements and counts them. Note that it
   has local state. *)

val count [a] (pipe : pipe_t a ack, ack : =ack)  : () =
  let c = newref 0 in
  preserving c @ ref int while true do begin
    let x = channel ::receive [packet a ack] pipe in
    (* We now have a permission to send on [ack]. *)
    assert ack @ outlet (bool | x @ a);
    incr c;
    tube ::send [(bool | x @ a)] (ack, true)
  end
  (* Stupid. This point in the code is unreachable. *)

(* Now, combine the producer and consumer. We cheat by running the producer
   on the current thread. If we spawned a new thread, we would have to
   transmit [t @ tree a] to it, and get it back afterwards, using [join]. *)

val test [a] (t : tree a)  : bool =
  (* [ack] must be allocated first, because the type of [pipe] refers to it! *)
  let ack = tube ::new() in
  let pipe = channel ::new() in
  thread ::spawn (fun ()  : () =
    count [a] (pipe, ack)
      (* Interesting example of a type application that is required. *)
  );
  iter (t, pipe, ack)
```

```
(* ----------------------------------------------------------------------------- *)

(*
Local Variables:
compile-command: "../mezzo tree-coroutine.mz"
End:
*)
```

A.5.2 channel.mzi

```
(* The type [channel a] describes a channel that can be used to send and
   receive messages of type [a]. This type is duplicable, so there can be an
   arbitrary number of senders and receivers. The type [a] need not be
   duplicable, so a message can imply a transfer of ownership. *)

abstract channel a
fact duplicable (channel a)

val new :     [a] () -> channel a
val send :    [a] (channel a, consumes a) -> ()
val receive : [a] channel a -> a
```

A.5.3 tube.mzi

```
(* A tube is a channel that is shared by only two participants.
   This allows messages of heterogeneous types to be sent over the
   tube. *)

(* A tube is initially inert. *)

abstract inert
fact exclusive inert

val new : () -> inert

(* A tube can be prepared for a single message exchange. The
   type of the message is fixed at this point. This gives rise
   to two dual permissions, for sending and for receiving. The
   two ends of the tube are known as the outlet and inlet. This
   operation does nothing at runtime. *)

abstract outlet -a
abstract inlet +a
```

```
val setup : [a] (consumes c : inert) -> (| c @ outlet a * c @ inlet a)

(* One permission allows sending, while the other allows receiving. *)

(* We adopt the convention that the receiver recovers the full
   ownership over the inert tube. This allows the tube to be
   re-used for further interactions. If [send] is asynchronous,
   then only the receiver can recover the full permission --
   it would be unsound for the sender to recover it, as it
   would then be possible to obtain two [inlet] permissions
   and use them in the wrong order. If [send] was synchronous,
   we could decide for either the receiver or the sender to
   receive full permission; these scenarios can probably be
   encoded in terms of asynchronous tubes anyway. *)

val send :    [a] (consumes outlet a, consumes a) -> ()
val receive : [a] (consumes c : inlet a) -> (a | c @ inert)

(* This notion of tube is less expressive than Lozes and Villard's
   automata, because the sender always loses all knowledge of the
   tube, and it is up to the receiver to establish a new convention
   (the type of the next message) and somehow transmit it to the
   other party. In other words, not much useful work can be done
   with just one tube; one always needs several tubes, or a tube
   and a standard channel. In contrast, Lozes and Villard are able
   to impose a complex protocol on a single channel. *)

(* If one is willing to perform a lot of ugly and costly dynamic
   checks, one might be able to encode any Lozes/Villard protocol
   as a system composed of two participants, a mediator, channels
   between each of the participants and the mediator, and tubes
   (distributed by the mediator) between the participants. The
   mediator would maintain the state of the automaton and set up
   the tubes for the next interaction. The participants would need
   to check at runtime that the state published by the mediator is
   indeed the state that they expect to be in. *)
```

A.6 vtube.mzi, vtube-iter.mz

A.6.1 vtube.mzi

```
(* These will be used as functions! *)
val send : unknown
val receive_ack : unknown
```

```
val receive : unknown
val send_ack : unknown

(* Sender's endpoint states *)
data send_ready (ep : term) a =
  S0 { contents : {p : perm} (|
      p * send @ (=ep, consumes x : a | consumes p) -> (| ep @ send_standby ep a x)
  )}
and send_standby (ep : term) a (x : term) =
  S1 { contents : {q : perm} (|
      q * receive_ack @ (=ep | consumes q) -> (| x @ a * ep @ send_ready ep a)
  )}

(* Receiver's endpoint states *)
data receive_ready (ep : term) a =
  R0 { contents : {p : perm} (|
      p * receive @ (=ep | consumes p) -> (x : a | ep @ receive_standby ep a x)
  )}
and receive_standby (ep : term) a (x : term) =
  R1 { contents : {q : perm} (|
      q * send_ack @ (=ep | consumes (x @ a * q)) -> (| ep @ receive_ready ep a)
  )}

(* Creates a new vtube, and returns it two endpoints in ready state *)
val new : [a] () -> (
  sender : send_ready sender a,
  receiver : receive_ready receiver a
)
```

A.6.2 vtube-iter.mz

```
data mutable tree a =
  Leaf
| Node { left : tree a; elem : a; right : tree a }

val rec producer [a] (t : tree a, e : vtube::send_ready e a): () =
  match t with
  | Leaf ->
    ()
  | Node { left; elem; right } ->
    (* Give (elem @ a) to the consumer *)
    vtube::send (e, elem);
    (* Get (elem @ a) back *)
    vtube::receive_ack e;
```

```
    producer (left, e);
    producer (right, e)
  end

(* A consumer that calls [f] on each element it receives *)
val rec consumer [a, p : perm] (
  e : vtube ::receive_ready e a,
  f : (a | p) -> ()
| p): () =

  let x = vtube ::receive e in
  (* x @ a *)
  f x;
  (* Send (x @ a) back to the producer *)
  vtube ::send_ack e;
  consumer (e, f)

val iter [a, p : perm] (
  t : tree a,
  f : (a | p) -> ()
| p): () =

  let (s, r) = vtube ::new () in
  let t =
    joinable_thread ::spawn
      (fun (| t @ tree a * s @ vtube ::send_ready s a): () =
         producer (t, s)
      )
  in
  consumer (r, f);
  match joinable_thread ::join t with
  | True -> ()
  | False -> fail
  end
```