Qui sème la fonction, récolte le tuyau typé

Didier Parigot, Bernard Serpette

Qui sème la fonction, récolte le tuyau typé

Didier Parigot, Bernard Paul Serpette

Inria Sophia-Antipolis - Méditerranée
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
First.Last@inria.fr
http://www-sop.inria.fr/members/First.Last

Résumé

Les applications de l'internet de demain vont devoir communiquer avec des objets de plus en plus complexes. Actuellement, cette communication s'effectue essentiellement à l'aide de protocoles de communication qui sont par nature des mécanismes mal typé. De plus, à chaque avancée technologique (sans fils, mobile...), de nouveaux types de protocoles de transport apparaissent.

L'objectif de l'article est de proposer un mécanisme de création de tuyau pour une communication bien typé et pour abstraire le protocole sous-jacent à la communication. L'idée de base, afin d'établir un tuyau, est de migrer du serveur vers le client une fonction qui se charge de mettre en place la structure du tuyau. Cette fonction permet au client de s'abstraire du protocole de communication. Ce tuyau, une fois établi, pourra aussi faire véhiculer des fonctions, cette fois-ci du client vers le serveur. Ces fonctions permettent au client d'exprimer l'échange d'information en termes d'expressions du langage source et donc d'assurer le typage de la communication. D'une autre manière, ces fonctions donnent les moyens au client de communiquer directement avec le serveur en cachant les détails du protocole puisque ces fonctions seront exécutées finalement sur le serveur.

Avec cette notion de tuyau, on définit un protocole comme étant un générateur de tuyau bien typé. Nous montrerons qu'il est possible d'établir des tuyaux de communication bien typé vers tous les objets atteignables sur un réseau.

1. Introduction

Le projet LogNet de l'Inria s'intéresse aux réseaux logiques : des structures distribuées où un programme peut s'intégrer, demander des ressources et activer ces ressources. Les entités participant au réseau communiquent entre elles pour assurer la fonctionnalité du réseau. Dans ce cadre, nous avons commencé le développement d'une bibliothèque Java, nommée Pinet, définissant l'interface d'un réseau en terme d'inscription et de requête, et des implémentations de cette interface. Une des implémentations visées est le système Arigatoni [1], mais nous visons aussi d'autres structures de réseaux comme celui de Chord [7]. Le réseau, quelque soit son implémentation, est utilisé, de manière abstraite, par le système orienté services de SmarTools [3] pour connecter les agents entre eux.

Le noyau de Pinet propose une structure abstraite de *tuyau* pour définir les communications entre les agents du réseau. Un tuyau est l'association d'un lecteur et d'un écrivain : les deux bouts du tuyau. L'implémentation du tuyau définit une structure qui véhiculera les données écrites par l'écrivain, vers le lecteur qui les lira. Il y a beaucoup de structure de tuyau, un fil reliant deux portes du micro-processeur est un tuyau véhiculant des bits, un registre, une variable C ou Scheme, une référence Caml, une queue, une pile, un couple de descripteurs de ficher, un flux TCP; toutes ces structures sont des tuyaux. Nos travaux portent essentiellement sur les flux connectant deux programmes sur deux machines différentes.

Dans cet article nous ne nous attarderons pas sur la structure des tuyaux, mais nous analyserons les lecteurs et les écrivains d'un point de vue du typage. Un lecteur est une fonction de type $unit \rightarrow \alpha$ et un écrivain une fonction de type $\beta \rightarrow unit$. Le rôle du tuyau, via sa structure, est d'unifier l'α et le β. Il va garantir, statiquement, que les valeurs écrites sont du même type que celles qui sont lues. Lorsque le tuyau est connecté sur une même mémoire, cette contrainte est facilement assuré par le vérificateur de type. Dans le cadre du calcul réparti, cette vérification est moins immédiate : le lecteur et l'écrivain sont souvent écrit dans des programmes indépendants (module serveur et module client par exemple). Pour résoudre ce problème, il suffit d'instancier le lecteur et l'écrivain dans un même programme et de transférer l'écrivain sur la machine devant faire les écritures. Malheureusement, les écrivains ne sont pas transférables d'une machine à l'autre, il manipule des informations liées à la structure du tuyau : descripteur de fichier, *socket*... Cette fois ci, le problème est résolu en transférant un générateur d'écrivain : une fonction qui créera l'écrivain sur la machine cible, i.e. celle qui ouvrira la *socket*. Ceci est la première raison qui nous a amené à considérer la migration de fonction : dans un même programme sont créés un lecteur et une fonction pouvant générer un écrivain, cette fonction migrera sur le réseau pour instancier l'écrivain au bon endroit.

D'un autre coté, on peut remarquer que derrière les protocoles de transport sur un réseau physique se cache généralement un interprète, simple pour HTTP avec les fonctions GET et POST; plus complexe pour le protocole Arigatoni. Quoi qu'il en soit, le principe reste que le client écrit une expression E qui sera décodée, i.e. interprétée, par le serveur. L'idée est de donner aux expressions tout le potentiel de celles d'un langage de haut niveau. L'expression E peut donc apparaître textuellement dans le code du client. Pour éviter que E soit exécutée sur le client, il faut utiliser une technique d'évaluation retardée, ou glaçon, qui consiste principalement à transformer E en une fonction $\lambda().E$. Rajouter un paramètre formel à la fonction permet d'abstraire le serveur du coté client, si E doit s'exécuter sur un serveur s dont on désire certaines fonctionnalités, i.e. s est libre dans E, alors $\lambda s.E$ est une fonction que l'on peut essayer de typer dans un programme client et qui peut-être écrite dans un tuyau vers le serveur s pour être auto-appliquée. Ceci nous a encore amené à considérer la migration des fonctions.

Cet article s'organise comme suit : dans un premier temps nous allons montrer, par un exemple Java, comment transmettre une fonction permettant d'abstraire l'objet serveur; dans une deuxième partie, nous analyserons le type Java des tuyaux pour leur donner une spécification fonctionnelle; dans une troisième partie nous définirons, en Caml, des générateurs de tuyaux pour construire une classe d'équivalence basée sur la relation "*est connecté à*"; puis nous reviendrons sur la création des tuyaux initiaux avant de conclure.

2. Le principe

Montrons, par un exemple simple, une communication typée et comment elle se différencie de RMI (*Java Remote Method Invocation*)[8] qui est la version orientée objet de RPC (*Remote Procedure Call*)[2]. L'exemple consiste à instancier un objet proposant un service, ici une simple méthode `hello` affichant un message à l'écran, et de montrer comment on active ce service à partir d'une autre machine. L'objet qui expose le service est communement appelé *le serveur* et l'objet qui requiert ce service *le client*.

En RMI, les services à exposer doivent passer par une interface.

```
public interface RmiI extends Remote {
    public void hello() throws RemoteException;
}
```

C'est cette interface qui sera visible par le client RMI. Le type `Remote`, super-classe de l'interface, cache la machinerie RMI qui fait que des objets seront accessibles de l'extérieur. Il convient maintenant

de définir une implémentation de cette interface.

```
public class RmiS implements RmiI {
    public void hello() {
        System.out.println("Hello");
    }
}
```

Le même code sera pris pour Pinet, hormis le fait qu'il n'est pas nécessaire de passer par une interface. Il est indéniable que cette interface permet de cacher l'implémentation du service au client, mais il est des cas où cette implémentation n'a pas à être cachée, ce sont plus les données manipulées par le serveur qui font sa particularité et qui empêche le service d'être exécuté sur le client : l'accessibilité d'une base de données par exemple. Pinet peut aussi passer par une interface, par contre RMI ne peut pas passer par une classe.

```
public class PinetS {
    public void hello() {
        System.out.println("Hello");
    }
}
```

Le plus important est que, dans cette définition de la classe `PinetS`, ou dans celles des interfaces qu'elle pourrait implémenter, rien ne montre que les instances de cette classe pourront ou devront être accessibles de l'extérieur. Toute classe Java peut potentiellement exposer ses méthodes à l'extérieur.

Le serveur doit maintenant créer une intance et l'exposer à l'extérieur :

```
public static void main(String args[]) {
    publishRMI(args[0], new RmiS());               /* Publication RMI   */
    publishPinet(PinetS.class, new PinetS());      /* Publication Pinet */
}
```

Dans les deux cas, RMI et Pinet, on crée l'objet à exposer et on utilise un système de publication. Ici nous voyons la première différence qui fait que Pinet expose les objets sous une clé qui représente précisément le type de l'objet exposé, alors que RMI utilise une chaîne de caractères[1] ce qui a posteriori, du coté client, va nécessiter une vérification dynamique du type de l'objet exposé.

En Java, `T.class` représente l'objet "*classe de* `T`" et est de type `Class<T>`. Ainsi la signature de la fonction `publishPinet` est : `<T> void publishPinet(Class<T>, T)`. Selon la notation décrite dans Henry & al[4], `T.class` s'écrirait en Caml : `<<T>>` et la signature de `publishPinet` deviendrait : $\forall\alpha.\texttt{TRepr}(\alpha) \rightarrow \alpha \rightarrow unit$.

Du coté client, il faut, à partir de la clé, rechercher un objet susceptible d'activer le service et activer ce service de manière appropriée :

```
public static void main(String[] args) {
    ((RmiI) lookupRMI(args[0])).hello();      /* Recherche et activation RMI  */
    lookupPinet(PinetS.class).write(new Proc1<PinetS>() { /* Recherche Pinet  */
            public void call(PinetS s) {
                s.hello();                                /* Activation Pinet */
            }});
}
```

[1] L'expression *(publishRMI name obj)* est équivalente à *LocateRegistry.getRegistry().rebind(name, UnicastRemoteObject.exportObject(obj, 0))*

La seconde différence, entre RMI et Pinet, concerne le résultat de la recherche qui est, pour RMI[2], un objet fournissant virtuellement les services requis et, pour Pinet, un objet permettant de communiquer avec l'objet fournissant réellement les services requis. Derrière l'objet de type `RmiI`, reçu par le client, se trouve toute la machinerie de transport sur le réseau : le code de la méthode `hello` de l'objet obtenu par le client envoie, via le réseau, un message qui activera la véritable méthode sur le serveur. Tout ce code spécifique est généré par compilation (`rmic`) ou automatiquement depuis la version 1.5. Ce code est appelé *talon* ou *stub* et se trouve pour une partie sur le client, pour l'autre sur le serveur.

Pinet propose une solution sans génération de code talon. Le client demande une valeur associée au *type* du serveur recherché, le résultat de cette requête est le bout de ce que nous appellerons un *tuyau* sur lequel il est possible d'écrire une procédure, i.e. une fonction ne retournant pas de résultat. Les procédures écrites dans le tuyau seront exécutées par la machine du serveur et recevront ce serveur en paramètre. Ainsi, l'expression `s.hello()` est écrite dans le programme client mais sera interprétée par le serveur.

Pour Pinet, la verbosité de code pour activer le service s'explique par le fait que Java n'a pas la notion de fonction anonyme. Le code "`new Proc1<PinetS>() {public void call XXX}`" devrait se réécrire, ou du moins doit se lire comme : `Lambda XXX`. Ainsi le code du client pourrait se réécrire, pour Scheme, en `(write (lookupPinet PinetS) (lambda (s) (hello s)))` ou, pour Caml, en `(write (lookupPinet <<PinetS>>) (fun (s) -> (hello s)))`.

Nous allons maintenant décrire la structure des tuyaux.

3. Le type des tuyaux

Dans l'exemple vu précédemment, nous n'avons qu'une partie des tuyaux : le résultat d'une recherche sur le réseau est le bout d'un tuyau sur lequel on peut écrire. L'autre bout du tuyau a été construit par le serveur au moment de la publication du service. Ainsi, un tuyau contient deux bouts, d'un coté on écrit/envoie, de l'autre on lit/reçoit. La seule contrainte que l'on voudrait imposer est que, pour un même tuyau, les valeurs écrites et lues soient de même type. C'est une contrainte assez forte que l'on retrouve dans les langages typés sous le fait que l'on ne peut pas avoir de structures répétitives (liste, vecteur, ensemble ...) hétérogènes, i.e. une liste contenant indifféremment des entiers et des fonctions. Mais, derrière cette restriction, on a l'assurance que le programme ne peut pas échouer du fait qu'une valeur d'un mauvais type ait été écrite dans un tuyau.

En Java, les lecteurs et les écrivains sur les tuyaux sont décrits par des classes abstraites paramétrées par le type des valeurs circulant dans le tuyau :

```
public abstract class InFlow<T> {
    public abstract T read();
}

public abstract class OutFlow<T> {
    public abstract void write(T value);
}
```

Comme ces deux classes ne proposent qu'une seule méthode et n'ont pas de données, elles sont équivalentes à de simples fonctions. Un lecteur est donc une fonction de type $unit \rightarrow \alpha$ et un écrivain une fonction de type $\alpha \rightarrow unit$.

Les lecteurs et écrivains sont certainement des objets encapsulant des données spécifiques au transport : la *socket* résultant d'une connexion TCP par exemple. Ces données n'ont de sens que pour

[2] *lookupRMI* est un alias de *java.rmi.Naming.lookup*

la machine qui les possède, elles ne sont pas transmissibles, on dit alors qu'elles sont *non sérialisables*. Par contre, ces lecteurs et écrivains sont construits à partir de valeurs plus simples (nom de machine, numéro de port ...) qui sont sérialisables. Il est donc possible d'encapsuler ces valeurs simples dans des objets qui vont circuler sur le réseau et pouvoir générer les écrivains à l'endroit souhaité. En Java, un générateur d'écrivains est encore décrit par une classe abstraite paramétrés par le type des valeurs circulant dans le tuyau :

```
public abstract class OutFlowGenerator<T> implements Serializable {
    public abstract OutFlow<T> generate();
}
```

Ici, on mentionne explicitement, par le mot clé `Serializable`, le fait que les générateurs d'écrivains peuvent circuler sur le réseau. De la même manière que pour les lecteurs et les écrivains, les générateurs d'écrivains sont équivalents à une simple fonction de type $unit \rightarrow OutFlow(\alpha)$, soit : $unit \rightarrow \alpha \rightarrow unit$.

L'étape suivante consiste à associer un lecteur avec un générateur d'écrivain de même type, ce qui donne en Java :

```
public class EndPointFlow<T> {
    public InFlow<T> inFlow;
    public OutFlowGenerator<T> outGenerator;
}
```

Ici la classe ne définit que deux champs sans implémenter de méthodes. Cette structure est donc un produit cartésien : $Inflow(\alpha) * OutFlowGenerator(\alpha)$, soit le type $(unit \rightarrow \alpha) * (unit \rightarrow \alpha \rightarrow unit)$. C'est ce type qui permet d'unifier le type paramétique du lecteur avec celui de l'écrivain, i.e. le même α. Ce produit cartésien, contenant directement un lecteur, n'est pas sérialisable. Pour ce faire, il suffit de définit un générateur de produit cartésien :

```
public abstract class Protocol implements Serializable {
   public abstract <T> EndPointFlow<T> generate();
}
```

Avec le même principe que précédemment, cette classe est équivalente au type : $unit \rightarrow (unit \rightarrow \alpha) * (unit \rightarrow \alpha \rightarrow unit)$. Toute fonction ayant ce type sera considérée comme un protocole. Nous allons maintenant donner deux exemples de protocoles, l'un en OCaml, l'autre en Java.

3.1. Exemples de protocoles

Voici un protocole définit en Objective Caml:

```
let qprot () =
   let q = Queue.create () in
      (fun () -> (Queue.pop q)),
      (fun () -> (fun (v) -> (Queue.push v q)))
```

Ce protocol n'est pas très intéressant du fait qu'il ne marche qu'en mémoire partagé et que la lecture n'est pas bloquante sur une queue vide, mais l'important est que la fonction a bien le type des protocoles. On voit ici le principe général des protocoles, qui, à leur activation, génère la structure du tuyau, ici une queue, qui sera utilisée par le lecteur, `(fun () -> (Queue.pop q))`, et par l'écrivain,

`(fun (v) -> (Queue.push v q))`. Par contre, la structure du tuyau, i.e. la queue, n'apparaît pas dans le type des protocoles.

Dans cette définition du protocole `qprot`, on ne voit pas la nécessité du générateur d'écrivains. Ici, l'écrivain `(fun (v) ...)` est sérialisable. Généralement, les écrivains ne sont pas sérialisables, voici une partie du code Java lié au protocole TCP :

```
public class Tcp extends Protocol {
    public <T> EndPointFlow<T> generate() {
        ServerSocket serversocket = new ServerSocket(0);
        return(new EndPointFlow<T>(
            new TcpSeqInFlow<T>(serversocket),
            new TcpOutFlowGenerator<T>(InetAddress.getLocalHost(),
                                       serversocket.getLocalPort())));
}   }

class TcpOutFlowGenerator<T> extends OutFlowGenerator<T> {
    private InetAddress where;
    private int port;
    public OutFlow<T> generate() {
        return(new TcpOutFlow<T>(new Socket(where, port).getOutputStream()));
}   }

class TcpOutFlow<T> extends OutFlow<T> {
    private ObjectOutputStream out;
    public void write(T value) {
        out.writeObject(value);
        out.flush();
}   }
```

On voit ici que le générateur d'écrivain `TcpOutFlowGenerator` ne contient que des valeurs sérialisables, à savoir l'adresse d'une machine et un port. L'écrivain effectif `TcpOutFlow` sera construit à partir de ces valeurs sérialisables. L'écrivain sera opérationnel là où le générateur aura été transmis et activé.

Pour en revenir au type des protocoles, il n'est pas indispensable que le lecteur et le générateur d'écrivain aient le même type polymorphe. Seuls les protocoles liés au transport physique sur le réseau doivent avoir cette particularité. Par exemple, si on cherche à cryptographier les valeurs transmises sur le réseau connaissant un protocole pour les valeurs cryptographiées, une fonction de cryptographie et son inverse, voici une fonction qui calcule le nouveau protocole :

```
let secure prot crypt uncrypt =
   let (read,writer)=(prot ()) in
      (fun () -> uncrypt (read ())),
      (fun () -> (fun v -> ((writer ()) (crypt v))))
```

Cette fonction `secure` crypte les valeurs avant de les donner à l'écrivain et décrypte les valeurs lues par le lecteur. Cette fonction est très polymorphe, son type est :

$(unit \to (unit \to \alpha) * (unit \to \beta \to \gamma)) \to (\delta \to \beta) \to (\alpha \to \epsilon) \to (unit \to \epsilon) * (unit \to \delta \to \gamma)$

Si on la spécialise avec un protocole de base comme `qprot` cela permet d'unifier α et β, donc le type de `secure qprote` est :

$(\alpha \to \beta) \to (\beta \to \gamma) \to (unit \to \gamma) * (unit \to \alpha \to unit)$

Et l'on obtient bien comme contrainte que la sortie de la fonction de cryptographie, β, doit correspondre à l'entrée de son inverse. Par contre, le lecteur et le générateur d'écrivains du résultat ne sont plus en relation : on lit des valeurs de type γ et on écrira des valeurs de type α.

Nous avons vu que les tuyaux pouvaient être vus comme des objets fonctionnels et qu'ils étaient typables. Nous allons voir maintenant comment, en utilisant des tuyaux faisant transiter des fonctions, on peut créer de nouveaux tuyaux bien typés.

4. Génération de tuyaux

Dans cette section, nous allons présenter un ensemble de générateurs de tuyaux. Toutes ces fonctionnalités prennent en argument (paramètre `out`) un écrivain de procédure sur un type `T`, par exemple le résultat de l'appel `lookupPinet(T.class)` que nous avons vu dans le premier exemple. Selon les types fonctionnels que nous avons vus précédemment, cet écrivain aura le type $(\alpha \rightarrow unit) \rightarrow unit$.

La première primitive indispensable est de pouvoir activer une *fonction* à distance plutôt qu'une *procédure*. Il faut donc faire revenir vers le client la valeur produite par la fonction. Si `out` est un écrivain de procédure ($(\alpha \rightarrow unit) \rightarrow unit$), si `f` est une fonction conforme à l'écrivain ($\alpha \rightarrow \beta$) et si `prot` est un protocole qui lit et écrit des valeurs de type β, alors la fonction suivante active `f` via `out` et retourne un lecteur du protocole `prot` avec lequel on peut lire la valeur calculée par `f`.

```
let detach prot out f  =
   let (read,writer)=(prot ()) in
      (out (fun (s) -> ((writer ()) (f s))));
      read
```

Dans un premier temps, on crée sur le client le lecteur (`read`) et le générateur d'écrivain (`writer`) pour la valeur de retour. Puis on écrit sur le tuyau `out` une procédure qui sera activée sur le serveur `s`. Cette procédure génère l'écrivain (`writer ()`), appelle la fonction `f` avec le serveur en argument (`(f s)`) et transmet le résultat de cet appel via l'écrivain. Ce résultat pourra être lu par le lecteur `read` qui est retourné par la fonction `detach`.

Comme la valeur de la variable `writer` est fermée dans la procédure qui est transmise via `out`, cette valeur transitera en même temps que la procédure. Ceci montre la nécessité que les générateurs d'écrivains soient sérialisables. De la même manière, la valeur fonctionnelle `f` doit aussi être sérialisable, tout autant que la valeur produite par `f`.

À titre d'indication, cette fonction `detach` nécessite trois fois plus de lignes pour être écrite en Java...

Dans la terminologie du calcul concurrent, la fonction `detach` engendre un appel asynchrone : elle n'attend pas le résultat de l'application effectuée à distance. La synchronisation se fait par activation du lecteur. Voici comment définir un appel synchrone.

```
let call prot out f = ((detach prot out f) ())
```

Cette fonction est très polymorphe, mais si on la spécialise avec un protocole et un écrivain (on prendra `write = ((snd (qprot ())) ())`), alors le type de `call qprot write` a exactement le type que l'on souhaite : $(\alpha \rightarrow \beta) \rightarrow \beta$.

On peut généraliser cette fonction par une fonction de symétrie qui, connaissant un serveur via un écrivain `out`, peux se faire connaître de ce serveur via un nouvel écrivain :

```
let sym prot out connect =
```

```
   let (read,writer)=(prot ()) in
      (out (fun (s) -> (connect s (writer ()))));
      read
```

Les deux premiers arguments de la fonction `sym` ont le même rôle que ceux de `detach`, et la notion "*se faire connaître via un nouvel écrivain*" est concrétisée par le dernier argument, `connect`, qui est une fonction recevant en paramètre le serveur et l'écrivain. Ainsi la fonction `detach` peut se redéfinir :

```
let detach prot out f = sym prot out (fun s w -> (w (f s)));
```

De la même manière, on peut définir une fonction de transitivité qui, connaissant un serveur `a` via un écrivain `out`, sachant que `a` connaît un autre serveur `b`, construit directement un tuyau vers `b` dont l'écrivain sera rendu en résultat.

```
let trans prot out get connect repp =
   let (read,writer)=(repp ()) in
      (out (fun (a) -> ((get a) (fun (b) -> let (readb,writerb)=(prot ()) in
                                            (connect b readb);
                                            ((writer()) writerb) ))));
      ((read ()) ())
```

Le premier argument, `prot`, est le protocol du tuyau que l'on veut construire, il transitera, via les fermetures, sur le serveur `a` puis sur le serveur `b`. Le second, `out`, est un écrivain de procédure sur le serveur `a`, le troisième, `get`, est une fonction qui, à partir du serveur `a`, trouve un écrivain de procédure sur le serveur `b`. Le quatrième, `connect`, est une fonction activée sur le serveur `b` qui prévient ce dernier de l'existence du lecteur sur le tuyau que l'on a créé. Le dernier, `repp`, est un protocol pour créer un tuyau temporaire.

La fonction `trans`, crée un lecteur et un générateur d'écrivain temporaire. Ce générateur va transiter, via les fermetures, sur le serveur `b`. Puis elle écrit, via le tuyau `out`, une procédure qui sera activée sur `a`. Cette procédure recherche, via la fonction `get`, un tuyau sur `b` et écrit sur ce dernier une nouvelle proédure. Celle-ci, une fois sur `b`, active le protocole pour obtenir un lecteur et un générateur d'écrivain (`readb` et `writerb`). Le serveur `b` est avertit de l'existence du lecteur via la fonction `connect` et transmet le générateur d'écrivain via le tuyau temporaire. La fonction `trans`, une fois la procédure transmise, sait qu'elle peut lire, sur le tuyau temporaire, un générateur d'écrivain, ce générateur est activé pour obtenir l'écrivain final : `((read ()) ())`.

Si la fonction `get` fait que le serveur `a` coincide avec le serveur `b`, i.e. `get = fun a f -> (f a)`, alors `trans` se comporte comme un duplicateur de tuyau. Néanmoins, cela permet de spécifier le protocole du tuyau dupliqué.

Avec cette notion de symétrie et de transitivité, en y ajoutant une réfléxivité évidente, la relation *connaître via un écrivain de fonctions* est une classe d'équivalence. Ainsi, on a la bonne intuition que tout ce qui peut être atteignable par le réseau peut se connecter directement par un tuyau typé.

Nous avons montré, qu'à partir de tuyaux typés, il est possible de créer d'autres tuyaux typés. Il reste à initialiser le processus en générant un premier tuyau. Nous allons donc rentrer un peu plus dans les détails des fonctions `publishPinet` et `lookupPinet` introduitent dans notre exemple.

5. Le grain est livré

Comment un client peut-il entrer en communication avec un serveur s'ils ne sont pas dans la même classe d'équivalence vue précédemment ? La première solution, hautement non typable, est

la convention : "**on** *sait que sur la machine* **www.inria.fr** *il existe un lecteur de type* **TCP** *sur le port* **80** *lisant du* **HTTP**". Ici *HTTP* fait référence au type des données qui vont circuler dans le tuyau, *TCP* et *80* sont plutôt en adéquation avec la structure du tuyau. Il y a la couleur des tuyaux et la couleur du liquide qui circule dedans. En regardant le fichier `/etc/services`, on serait tenté de croire qu'il existe une relation entre les deux couleurs. Mais ce n'est qu'une convention, de fait on peut utiliser le port 80 pour tenter de passer au travers d'un pare-feux et de faire transiter autre chose que du HTTP. Néanmoins, l'intérêt de cette convention est que l'Inria n'a pas à prévenir le monde entier qu'un serveur http a été installé sur sa machine.

Une solution, moins directe, consiste à faire intervenir une tierce personne pour établir la communication. Le serveur va publier, sur un site particulier, via un annuaire, la couleur de son tuyau d'entrée et la couleur de ce qu'il faut mettre dedans. La couleur du contenu peut-être implicite (annuaire mono-thématique comme dans le cadre de BitTorrent par exemple), seul l'adresse où se trouve le lecteur est importante (i.e. *www.inria.fr*), mais dans ce cas les tuyaux sont monomorphes et ne posent pas problèmes. Le client, de son coté, utilise l'annuaire pour construire l'écrivain avec les bonnes couleurs. La publication et la recherche se font au travers d'une clé, on publie un objet sous une certaine clé et on recherche un objet ayant été publié sous une certaine clé. Une particurité de Pinet est qu'il y a une relation de type, donc statiquement vérifiable, entre la clé et son contenu, le type Java de l'annuaire est :

```
Hashtable<Class<T>, CircList<OutFlowGenerator<Proc1<T>>>>
```

En Caml ce devrait être : `TRepr`$(\alpha) \rightarrow List(unit \rightarrow \alpha \rightarrow unit)$. Le fait que la liste soit circulaire est juste une optimisation pour assurer que les serveurs sont répartis de manière uniforme au fur et à mesure des requêtes. Il est important de remarquer que le type Java `OutFlowGenerator<Proc1<T>>` est proche du type du résultat de la fonction `lookupPinet` utilisée dans l'exemple du début de l'article. En fait, cette fonction va chercher un générateur d'écrivain sur le dictionnaire distant et active localement ce générateur pour produire l'écrivain.

Mais, comment font le client et le serveur pour rentrer en contact avec le propriétaire de l'annuaire ? La première solution, hautement non typable, est la convention : "**on** *sait que sur la machine* **ariwheels.inria.fr** *il existe un lecteur de type* **TCP** *lisant sur le port* **5359** *lisant des fonctions Java . . . "* etc. . . etc.

Doit-on croire que le problème a juste été repoussé et que l'on est en face du dilemme de la poule et de l'oeuf ? En fait, oui. De la même manière qu'un vérificateur de type ne pourra jamais prouver que les arguments de la fonction `main` sont bien des chaines de caractères[3]. Il faut accepter que tout programme démarre avec un ensemble, potentiellement vide si hors-réseau, de tuyaux typés. Néanmoins, à chaque fois que l'on repousse la création du tuyau originel, les types des valeurs devant circuler dans ce tuyau devient moins général.

6. Conclusion

Nous avons présenté le noyau de la librairie Pinet permettant de typer statiquement les valeurs circulant sur un réseau. Nous avons montré que cette propriété a été atteinte en admettant qu'il est possible de faire migrer du code, i.e. que les fonctions pouvaient transiter d'une machine à l'autre. Java a cette particularité, c'est une nécessité du fait que la principale entité que traite Java est l'objet, et sachant qu'un objet est l'encapsulation de données et de fonctions, i.e. une fermeture avec plusieurs points d'entrée, ne pas savoir envoyer une fonction sur le réseau reviendrait à reconnaître ne pas savoir envoyer des objets sur le réseau.

[3] il ne faut jamais dire jamais, Compcert[6], après avoir prouvé les compilateurs, pourrait s'attaquer à la preuve des systèmes d'exploitation

La notion de tuyaux typés est validée par le fait que, sur l'ensemble du code de la librairie, ne subsiste plus qu'un seul test de type dynamique. Il se trouve dans l'implémentation des lecteurs liés à un flux de caractères, la classe `InputStream` de Java, et vérifie le résultat d'une dé-sérialisation (méthode `readObject` de Java correspondant à la fonction `Marshal.from_channel` d'Objective Caml).

Pinet utilise de manière intense le polymorphisme générique hérité de Pizza[5]. En nous ramenant à des types fonctionnels nous avons pu exprimer les générateurs de tuyaux en Caml. Sans considérer la concision gagnée, cela nous a permis d'observer que les fonctions écrites originellement en Java étaient plus générales que prévu. Mais il aurait été fastidieux d'avoir à écrire ces types polymorphes en Java. Une des conclusions est donc que le fait d'obliger le programmeur à typer ses variables peut le conduire à minimiser la puissance des fonctions qu'il écrit.

Avoir des tuyaux typés implique qu'ils sont homogènes, i.e. que l'on ne peut pas écrire, par exemple, un entier puis un caractère. Ceci peut amener à créer plus de tuyaux que nécessaire. La structure d'un tuyau TCP est assez coûteuse à mettre en place. Dans ce contexte, le projet LogNet s'intéresse à une notion de *canal*, structure physique permettant une communication entre deux machines, pouvant contenir des structures plus légères, les *canaux virtuels*. Les tuyaux que nous avons décrits dans cet article peuvent être considérés comme des canaux virtuels. Il nous reste encore à essayer de typer les canaux.

Si au commencement de l'élaboration de la librairie Pinet, le sûreté par le typage était anecdotique et relevait du jeu : il aurait été certainement plus facile de rajouter des tests de type dynamiques là où le compilateur le demandait. *In fine*, l'effort fournit pour éliminer ces tests de type dynamiques nous semble profitable et nous a permis de structurer la librairie de manière plus abstraite. Enfin, cela nous a amené à des reflexions que nous avons essayé de partager dans cet article.

Bibliographie

[1] Didier Benza, Michel Cosnard, Luigi Liquori, and Marc Vesin. Arigatoni: A simple programmable overlay network. In *JVA '06: Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*, pages 82–91, Washington, DC, USA, 2006. IEEE Computer Society.

[2] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[3] Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. Un modèle abstrait de composants adaptables. *revue TSI, Composants et adaptabilité*, 23(2), 2004.

[4] Grégoire Henry, Michel Mauny, and Emmanuel Chailloux. Typer la dé-sérialisation sans sérialiser les types. *Journées francophones des langages applicatifs*, pages 133–146, January 2006.

[5] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *In Proc. 24th ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM, 1997.

[6] Le projet compcert. ANR project. `http://compcert.inria.fr/`.

[7] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.

[8] Sun. Java remote method invocation - distributed computing for java. white paper, 1998. `http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp`.