CARI'10

Un protocole de fertilisation croisée d'un langage fonctionnel et d'un langage objet: application à la mise en oeuvre d'un prototype d'éditeur coopératif asynchrone

Maurice TCHOUPÉ TCHENDJI*

* Département de Maths-Informatique
Faculté des Sciences
Université de Dschang
BP 67, Dschang-Cameroun
ttchoupe@yahoo.fr

RÉSUMÉ. La fertilisation croisée est une technique permettant de mettre en commun des compétences et des ressources d'au moins deux secteurs d'activité afin d'en tirer le meilleur de chaque. Dans ce papier, nous présentons un protocole de programmation basé sur la fertilisation croisée de deux langages de programmation (Haskell et Java) relevant de deux paradigmes de programmation différents: le paradigme fonctionnel et le paradigme objet. Cette mutualisation des points forts de chaque type de langage permet de développer des applications plus sûres, en un temps moindre, ayant un code fonctionnel concis, facilement compréhensible et donc, facilement maintenable par un tiers. Nous présentons la méta-architecture des applications développées suivant cette approche ainsi qu'une instanciation de celle-ci pour la mise en oeuvre d'un prototype d'éditeur coopératif asynchrone.

ABSTRACT. The cross-fertilization is a technique to pool expertise and resources of at least two sectors in order to make the best of each. In this paper, we present a protocol of programming based on cross-fertilization of two programming languages (Haskell and Java) under two different programming paradigms: the functional paradigm and the object paradigm. This pooling of the strengths of each type of language permit to develop more secure applications in a shorter time, with functional code concise, easily understandable and thus, easily maintainable by one third. We present the meta-architecture of applications developed following this approach and an instantiation of it for the implementation of a prototype of an asynchronous collaborative editor.

MOTS-CLÉS : Fertilisation croisée, Programmation fonctionnelle, Programmation Objet, Edition coopérative, Parseurs, Evaluation paresseuse, XML.

KEYWORDS : Cross-fertilization, Functional Programming, Object Programming, Cooperative Edition, Parsers, Lazy evaluation, XML.

1. Introduction

Depuis la création dans les années 50 de la première version de LISP (langage de programmation fonctionnelle utilisant le lambda (λ) calcul de Church) par MacCarthy, les membres de la communauté de programmation fonctionnelle n'ont jamais eu de cesse de travailler avec entrain dans l'espoir qu'un jour ces langages aient toutes les qualités requises afin d'être utilisés pour le développement des applications industrielles et commerciales. Paradoxalement, malgré les caractéristiques intéressantes dont jouissent ces langages (la plus intéressante est sûrement leur fondement mathématique - λ calcul - qui permet d'y écrire des programmes dont on peut aisément prouver l'exactitude) ils ont longtemps peiné à être adoptés par les programmeurs pour le développement logiciel.

Dans les années 90, Hudak et al. dans [1] tout en constatant qu'il existe déjà des avancées en matière d'utilisation de ces types de langages pour le développement des applications suggèrent qu'ils peuvent être utilisés avec davantage de succès (au détriment des langages impératifs classiques : c, java, ...) pour le prototypage des applications. Une telle utilisation procurera de nombreux avantages hérités du paradigme fonctionnel : temps de développement moindre (par rapport au temps de développement du même prototype dans un langage impératif [1, 2]), code concis, facile à comprendre et à en prouver l'exactitude... Une comparaison des prototypes de la même application réalisés avec Haskell, Ada et C++ respectivement, comparaison portant sur le nombre de lignes de code, le temps mis pour le développement, la production de la documentation, établit clairement la suprématie de Haskell sur les autres langages [1]. De même, dans [3] on a un exemple d'algorithme classique, le "QuickSort", qui est implémenté en C++ et en Haskell. Ici également, on note que la version Haskell (fonctionnelle) du dit algorithme est à la fois plus naturelle à écrire qu'à comprendre.

Le paradigme objet quant à lui a émergé au cours des années 70-80 et a été largement adopté par les développeurs de par les avantages qu'il procure : modularité, encapsulation, polymorphisme, héritage, ... Il s'en est suivi le développement de plusieurs langages orientés objets comme *SmallTalk, C++,* ...et surtout *Java* [7] qui est largement utilisé de nos jours. Notons toutefois que les concepts qui ont séduit les adeptes du paradigme objet ne sont pas inhérents à ce seul type de programmation. On les trouve aussi dans d'autres types de programmations notamment dans le type fonctionnel qui possède en plus d'autres concepts intéressants : évaluation paresseuse, haut niveau d'abstraction ...qui permettent d'écrire des outils logiciels exacts, robustes, extensibles, réutilisables, portables, efficients et ce, en écrivant un code concis et naturel (ie proche de la solution algorithmique élaborée). Ces paradigmes qui ont en partages des concepts (intéressants) similaires sont de bons candidats pour une fertilisation croisée.

Dans ce papier, loin de préconiser l'utilisation des langages appartenant à tel ou tel autre paradigme pour le prototypage ou le développement des applications, nous proposons une approche hybride dite de *fertilisation croisée* qui prône l'utilisation de plusieurs langages (appartenant à des paradigmes éventuellement différents) pour le développement ou le prototypage des applications en vue d'en tirer le meilleur de chaque. Nous nous intéressons essentiellement dans ce qui suit et ce, sans perte de généralité, aux paradigmes fonctionnel et objet à travers les langages Haskell [6] et Java [7]. Plus spécifiquement, dans ce papier, nous soutenons l'idée selon laquelle on peut avantageusement tirer le meilleur de ces paradigmes en utilisant des langages appartenant à chacun d'eux pour bâtir une application. En fait, sachant que le principal handicap en terme de prise en main

par des programmeurs habitués au style de programmation procédurale et désirant expérimenter l'approche fonctionnelle est non seulement le style de programmation qui a cours ici (pas de notion d'état) mais aussi et surtout la mise en oeuvre des interfaces graphiques[1] (le traitement des effets de bord), nous préconisons l'utilisation d'un langage fonctionnel (Haskell) pour mettre en oeuvre la partie métier de l'application et un langage objet (Java) pour les autres préoccupations : interface utilisateur, accès aux bases de données, sérialisation, ... Nous sommes confortés dans notre approche par l'usage aisé que nous en avons fait pour l'implémentation d'un prototype d'éditeur coopératif asynchrone dont les grandes lignes de développement ainsi que quelques captures d'écran sont données dans ce papier.

La suite de ce papier est organisée comme suit : notre approche de mise en oeuvre d'une fertilisation croisée Haskell-Java ainsi qu'une méta-architecture des applications développées suivant cette approche sont présentées dans la section 2. Nous présentons ensuite (section 3) la problématique générale de l'édition coopérative asynchrone puis, un prototype d'éditeur coopératif asynchrone mis en oeuvre suivant notre approche. La section 4 est consacrée à la conclusion. Nous y faisons un bilan critique de notre travail et présentons quelques pistes pour des travaux futurs. Enfin, en annexe, nous donnons le code de quelques classes et fonctions ayant servi à la mise en oeuvre d'un générateur de nombres premiers. Générateur bâti suivant notre protocole de fertilisation croisée Haskell-Java. Ces fonctions sont données afin qu'un lecteur désirant expérimenter notre approche de la fertilisation croisée Haskell-Java puisse s'en inspirer (surtout s'il n'est pas un habitué de XML) pour écrire certains modules de son application.

2. Fertilisation croisée Haskell-Java

Cette section présente notre approche de la fertilisation croisée entre Haskell et Java. Cette approche est basée essentiellement sur la possibilité qu'on a de pouvoir appeler un programme exécutable - Haskell - à l'intérieur d'un programme Java[2]. Dans ce qui suit, après avoir présenté une méta-architecture pour les applications à réaliser suivant cette approche, nous montrons comment elle peut-être instanciée en une application concrète.

2.1. Une méta-architecture pour les programmes développés par fertilisation croisée Haskell-Java

Le protocole de programmation des programmes développés au moyen de la fertilisation croisée Haskel-Java que nous proposons est schématisé sur la figure 1 : c'est un modèle inspiré des RPC (Remote Procedure Call)[3]. Ce protocole peut-être présenté simplement comme suit : partant de deux programmes, l'un (interactif) écrit en Java et l'autre

1. Il existe beaucoup de bibliothèques graphiques pour Haskell (FranTk, Fruit, wxHaskell, ...) mais, malheureusement aucune n'émerge comme standard et toutes sont plus ou moins incomplètes [8]. Leurs prise en main est dès lors assez difficile et leurs utilisations assez limitées.

2. Notons que la même chose est possible à partir du langage Haskell ie. appeler un exécutable - Java - à partir d'un programme Haskell [13].

3. Un RPC correspond généralement à deux IPCs : un message est envoyé par le "client" vers le "serveur" contenant l'identifiant de la procédure à appeler et les paramètres (encodés selon un protocole précis), puis un message est envoyé de la part du "serveur" vers le "client" avec la valeur de retour de la fonction, ou des informations en cas d'éventuelles erreurs.

(en mode batch) écrit en Haskell, l'idée consiste à insérer dans le programme Java un ou plusieurs *"points d'entrées"* à partir desquelles on invoquera une ou plusieurs routines (programmes) Haskell en leurs transmettant des valeurs idoines correspondant à leurs arguments. Puis de s'assurer que le programme Haskell (appelé) s'exécute et retourne un résultat encapsulé dans un message qui est *intercepté* dans le programme Java (appellant).

Ainsi présenté, le seul véritable problème qui subsiste est celui relatif au format dans lequel les paramètres (données et résultats) doivent être codés. Nous préconisons pour cela un format à la XML, inspiré de celui utilisé dans la définition et l'appel de fonctions dans le langage XSLT [9]. Plus précisément, il s'agit d'encapsuler (après d'éventuelles conversions) les différentes valeurs des arguments d'appels dans un document XML bien formé et éventuellement valide vis-à-vis d'un certain modèle de document. On procède de même pour le résultat issu de l'exécution du programme Haskell. Pour la mise en oeuvre effective de cette infrastructure, on doit avoir dans chacun des programmes Java et Haskell d'un utilitaire spécifique - nous l'appelons *transducteur*[4] - qui effectue les tâches suivantes :

1) A l'invocation du programme Haskell, le transducteur (*codeur*) réalise préalablement un codage (ré-écriture) des différents arguments de l'appel vers le format (XML) convenable.

2) Dans le programme Haskell, le transducteur (*parseur*) analyse le document XML encapsulant les différents arguments, les extrait et convertit chacun dans le type (de base) convenable utilisé dans le programme Haskell.

3) A la fin de l'exécution du programme Haskell, le transducteur (*codeur*) réalise un codage du résultat de l'exécution dans le format convenable et l'encapsule dans un document XML à retourner au programme Java.

4) A la réception du résultat de l'exécution du programme Haskell dans le programme Java, le transducteur (*parseur*) analyse le document résultat qui lui est acheminé pour y extraire les données qu'il contient. Ces données sont ensuite converties dans le type (Java) convenable utilisé dans la suite du programme. On en déduit (figure 1) une méta-architecture pour les programmes à développer suivant ce protocole.

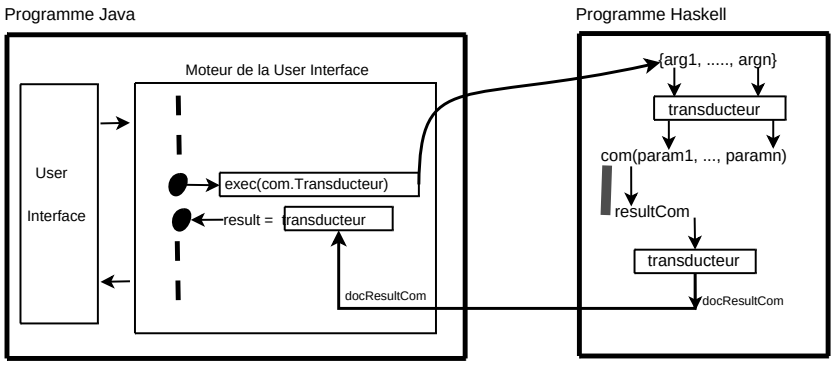

Figure 1. *Méta-architecture des applications à développer par fertilisation croisée Haskel-Java*

4. Le transducteur ici est perçu comme une routine de transformation d'une syntaxe dans une autre. Il opère par appel à deux sous-routines : le *codeur* pour la codification ie. la fabrication d'un message (un fichier XML) et le *parseur* pour la découverte (extraction de l'information) du message.

La mise en oeuvre des différents transducteurs ne pose pas de problème particulier. On peut par exemple soit les écrire de but en blanc, auquel cas, on pourra utiliser les techniques et/ou les bibliothèques existantes [4, 5] ou utiliser à bon escient des outils existants comme XSLT[9]. Un exemple de *parseur* - l'une des sous-routines du *transducteur* - écrit en Java et utilisant une feuille de style XSLT est donné dans la section 6.1 (annexe).

Le type des documents XML permettant de fabriquer un message en vue de l'invocation d'une fonction écrite dans l'autre langage est donné par la DTD suivante :

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<!-- DTD Commande caractérise une commande et ses arguments -->
<!ELEMENT Commande (NomCommande, Arguments?) >
<!ELEMENT NomCommande (#PCDATA)>
<!ATTLIST NomCommande  fichier CDATA #IMPLIED >

<!ELEMENT Arguments (Arg+)>
<!ELEMENT Arg (#PCDATA)
<!ATTLIST Arg  type CDATA #IMPLIED >

```

Le document suivant est un document XML conforme à la DTD précédente. Elle permet d'invoquer la fonction *nPremiers* du programme *"crible.exe"* - présenté dans l'exemple de la section suivante - en lui fournissant comme paramètre la valeur entière *"10"*.

```
<Commande>
   <NomCommande fichier = "crible.exe"> nPremier </NomCommande>
       <Arguments>
          <Arg type = "Integer"> 10 </Arg>
      </Arguments>
</Commande>
```

Le résultat fournit suite à l'invocation de cette fonction est le fichier XML suivant :

```
<Resultat reponseStructure = "Yes" typeElement = "int" separateur = ",">
   2,3,7,11,13,17,19,23,29
</Resultat>
```

La sémantique associée à ce fichier est celle-ci : le résultat du crible est une liste d'entiers, les éléments de cette liste sont séparés par des virgules. En fait, les documents retournés au programme appelant Java par le programme Haskell sont conformes au modèle de document donné par la DTD suivante :

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<!-- DTD Resultat caractérise le résultat de l'exécution d'un fichier Haskell -->
<!ELEMENT Resultat  (#PCDATA)>
<!ATTLIST Resultat
      reponseStructure (Yes | No) "No" #REQUIRED <!--la réponse est-elle structurée
                                du genre "Liste" ou est-elle d'un type primitif? -->
      typeElement (byte, short, int, long, float, double, char, boolean, String)
                                                          "String" #REQUIRED
      separateur CDATA #IMPLIED> <!--Séparateur d'élément utilisé dans le cas où
                                              la réponse est structurée -->
```

2.2. Mise en oeuvre et illustration

Nous présentons dans cette sous-section une instanciation du protocole en vue de la réalisation d'une application simple dont l'implantation utilise un programme Haskell et un programme Java. Le programme Haskell utilisé est une implantation de l'algorithme du crible d'Eratosthène permettant de déterminer les *n* premiers nombres premiers ; la valeur de *n* est fournie comme argument lors de l'exécution du programme. On notera ici l'utilisation des listes infinies en Haskell *([2..] ≡ liste infinie d'entiers à partir de 2, ie. 2,3,4, . . .)* qui, grace au mécanisme d'évaluation paresseuse ne cause pas de problème spécifique comme celui de la non termination du programme. Le programme Java quant à lui réalise l'interface - graphique - d'interaction avec l'utilisateur. Conformément au protocole présenté dans la section précédente, il recueille la donnée saisie par l'utilisateur, l'achemine au programme Haskell après l'avoir encapsulée dans un document XML, puis, recueille le résultat fourni par le programme Haskell, la *dépouille* pour y extraire le résultat et l'afficher. Dans ce qui suit, nous nous servons donc de l'algorithme du crible d'Eratosthène pour illustrer une instanciation de notre protocole sur un exemple simple. Pour cette fin, nous présentons successivement comment produire un exécutable Haskell puis comment exécuter un programme Haskell dans Java suivant notre approche de la fertilisation croisée Haskell-Java.

Production d'un exécutable Haskell

L'implémentation GHC [10] de Haskell est un compilateur. On peut donc y créer des programmes exécutables et les faire exécuter indépendamment du compilateur qui a servi à leurs générations. Un programme Haskell sous GHC est un fichier d'extension *.hs* semblable à celui ci-dessous[5] enregistré par exemple sous le nom *crible.hs*.

```
1  module Main where
2  import System.Environment
3  main = do
4            args <- getArgs --récupération des paramètres d'appel du programme
5            if (length args) /= 1 then do
6                     putStr("Usage: NomProgramme Argument")
7                      else do putStr((show (nPremiers (read (head args)))))
8  nPremiers n = if (n < 1) then do error " le parametre doit être positif!"
9                 else do take n (crible [2..])
10 --Implémentation de l'algorithme réalisant le crible d'Eratosthène
11 crible [] = []
12 crible (n:ns) = n:(crible [m | m <- ns, (m `mod` n) /= 0])
```

Pour compiler un tel programme (avec *GHC*) et créer un exécutable nommé *crible.exe*, il suffit de saisir la commande *"ghc –make -o crible crible.hs"*. Pour l'exécution, il suffit de saisir la commande *"crible x"* où *x* est l'argument de l'appel[6]. Par exemple, *crible 10* pour l'affichage des 10 premiers nombres premiers.

5. Ce programme est une implémentation du *crible d'Eratosthène* permettant de déterminer les *n* premiers nombres premiers ; la valeur de *n* est fournie comme argument lors de l'exécution du programme.

6. En fait, avec *GHC*, on peut créer des programmes qui utilisent des arguments de la ligne de commandes. Ces arguments sont accessibles dans le programme source par le truchement de la variable *args* (voir ligne *04* dans

Exécution d'un programme Haskell dans Java

Java permet de lancer un code exécutable (quelconque) à partir d'un code java, et donc, en particulier, du code obtenu à partir d'un programme Haskell. On peut procéder comme suit :

1) Créer l'exécutable Haskell comme présenté ci-dessus,

2) Rassembler les arguments d'appel de l'exécutable Haskell dans une variable (disons *cmd*) de type (Java) *[] String* telle que : *cmd [0]* contient la référence (chemin d'accès) du fichier exécutable Haskell et *cmd [1]* est un document XML encapsulant les paramètres d'appel de cet exécutable. En rappel, en adoptant la notation XPATH permettant d'écrire les chemins de localisation des noeuds dans un document XML, ce fichier est structuré comme suit (voir section 2.1) :

- */Commande/nomCommande/@fichier* contient la référence (chemin d'accès) du fichier exécutable Haskell et,

- */Commande/nomCommande* contient le nom de la commande (fonction) à exécuter.

- */Commande/Arguments/Arg* contient les différents arguments de l'appel.

3) Créer un *process* java pour lancer l'exécutable Haskell,

4) Rediriger convenablement la sortie (document XML) de l'exécutable Haskell de façon à la *récupérer* pour traitement dans le programme (Java) courant.

Le listing suivant donne quelques extraits d'un exemple de programme Java invoquant le programme Haskell conçu précédemment. La quasi-totalité du programme est donnée dans l'annexe 6.

```
1  //classe permettant de créer un processus pour y exécuter un programme externe
2  class ExecuteExternProgramInJava {
3     //passage par argument de la commande à lancer
4     static String StartCommand(String [] command) {
5       try {
6          //creation du processus
7          Process p = Runtime.getRuntime().exec(command);
8          InputStream in = p.getInputStream();
9            //on récupère le flux de sortie du programme
10         StringBuffer build = new StringBuffer();
11         char c = (char) in.read();
12         while (c != (char) -1) {
13             build.append(c);
14             c = (char) in.read();
15         }
16         String resultat = build.toString();
17         return resultat;
18      }
19      catch (Exception e) {
20        System.out.println("\n" + command + ": commande inconnu ");
21        return "";
```

le listing ci-dessus). La fonction *getArgs* est définie dans le module *System.Environment* et elle a pour type : *getArgs : : IO [Strings]*.

```
        }
    }
    }
...
//classe réalisant l'interface graphique
class HaskellDansJava implements ActionListener {
  ...
  public void actionPerformed(ActionEvent événement) {
    String[] cmd = new String[2];
    cmd[0] = "crible.exe" ;
         //voir l'annexe pour la classe "Traducteur"
    cmd[1] = Transducteur.codeur("crible.exe", "nPremier", "Integer 10");
    String resulHaskell = ExecuteExternProgramInJava.StartCommand(cmd);
       //chaine de caractères représentant la feuille de style XSLT
    String xsl = "<xsl:stylesheet version=\"2.0\" " +
                  "xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">" +
                  "<xsl:output method=\"text\"  />" +
                  "<xsl:template match=\"Resultat\"> "+
                     "<xsl:value-of select=\".\"/>"+
                  "</xsl:template>"+
                "</xsl:stylesheet>";
    resulHaskell = Transducteur.parseur(resulHaskell, xsl);
    resultat.setText(resulHaskell);
   }
  ...
}
```

L'exécution du programme précédent produit la sortie suivante (capture d'écrans) :

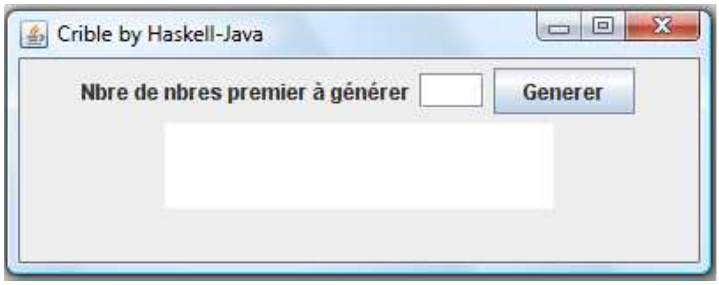

Figure 2. *Interface du programme réalisant le crible.*

3. TinyCE : un prototype d'éditeur coopératif asynchrone

Cette section est consacrée à la présentation de *TinyCE : a Tiny Cooperative Editor*, un prototype d'éditeur coopératif asynchrone développé suivant notre approche de fertilisation croisée Haskell-Java. Comme nous le verrons dans ce qui suit, il était difficile de

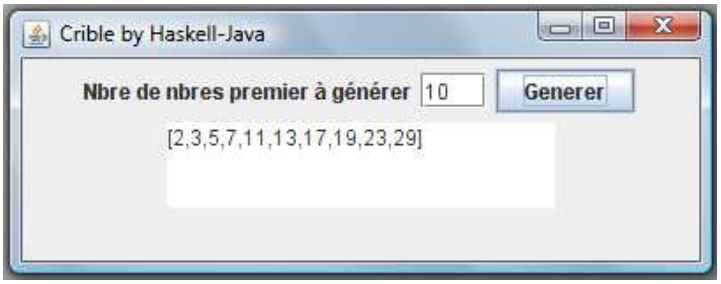

Figure 3. *Interface du programme après exécution du crible.*

développer ce prototype uniquement avec Haskell à cause des interfaces graphiques complexes de cet éditeur (voir figures 6 et 7), et quasi-impossible de le réaliser uniquement en Java à cause des données généralement infinies qu'on y manipule.

3.1. Édition coopérative et TinyCE

L'édition coopérative est un travail de groupe hiérarchiquement organisé qui fonctionne suivant un planning impliquant des délais et un partage des tâches. Nous avons proposé dans [11, 12] un modèle conceptuel pour l'édition coopérative asynchrone basé sur la notion de vue partielle (des documents). Nous avons utilisé les grammaires algébriques comme *modèle de documents*[7]. Un document t dans ce modèle est un *arbre de dérivation* pour une grammaire $\mathcal{G}$ donnée. Une vue partielle (aussi appelée réplicat partiel) du document t édité de façon coopérative correspond aux différentes sous-parties du document t contenant des objets qui sont pertinents pour un utilisateur particulier participant à l'édition coopérative. Plus précisément, c'est le document t dans lequel on a supprimé tous les noeuds étiquetés par certains symboles grammaticaux jugés non pertinents pour l'utilisateur considéré, tout en conservant la structure de sous-arbre. La figure 4 présente un document (au centre) ainsi que deux vues partielles de celui-ci obtenues en retenant comme objets pertinents les symboles $A,\ B$ pour la vue partielle de gauche et $A,\ C$ pour celle de droite. Chaque utilisateur participant à l'édition coopérative possède un réplicat

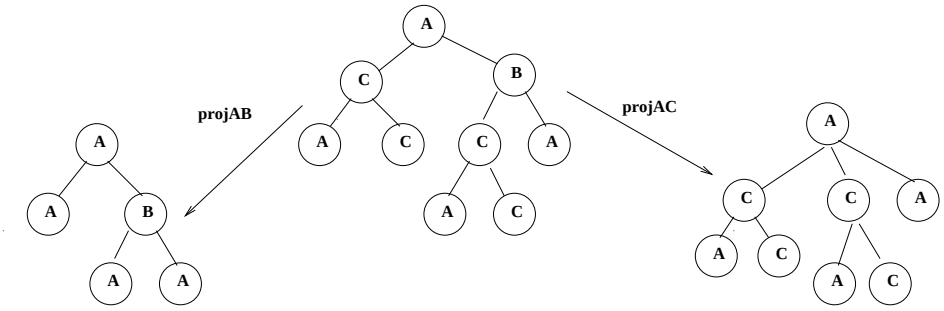

Figure 4. *Un document et deux vues partielles obtenues en retenant comme symboles pertinents respectivement $A,\ B$ et $A,\ C$.*

partiel sur lequel il agit (édite) de façon assynchrone. A un moment donné, pour obtenir un document t intégrant toutes les contributions des différents auteurs, il faut *fusionner*

7. Le choix des grammaires algébriques est motivé par le fait qu'elles sont plus générales que les DTDs de XML.

tous les réplicats partiels. Il existe malheureusement des situations où le nombre de documents t_f correspondant à la fusion des différents réplicats partiels est infini [11, 12] : d'où la nécessité d'utiliser pour cette opération des structures de données paresseuses.

Ci-dessous nous présentons brièvement comment l'implémentation de *TinyCE* - éditeur coopératif asynchrone utilisant cette notion de vue partielle - a été menée au moyen d'une fertilisation croisée Haskell-Java. Notons une fois de plus qu'il était difficile de réaliser ce prototype exclusivement avec *Java* à cause de la manipulation dans cet éditeur des données généralement infinies : par exemple, le résultat de l'opération de *projection inverse* [11] est généralemnt une liste infinie d'éléments et Java n'est pas un langage paresseux.

TinyCE est un éditeur WYSIWYG[8] permettant l'édition conviviale (édition graphique par simple utilisation de la souris) de façon coopérative et asynchrone de la structure abstraite (arbre de dérivation) des documents structurés. Il s'utilise en réseau suivant un modèle client-serveur. Son interface utilisateur (fig. 6 et fig. 7) offre à l'usager des facilités pour l'édition du modèle des documents (une grammaire), des vues qui lui sont associées, l'édition et la validation d'un document global ou d'un réplicat partiel de celui-ci suivant que l'utilisateur est sur le poste serveur[9] ou client. Bien plus, cette interface lui offre aussi des fonctionnalités lui permettant d'expérimenter les notions de *projection*, d'*expansion* et de *fusion* de documents [11, 12].

3.1.1. Architecture de TinyCE

TinyCE est constitué de trois composants principaux : l'interface utilisateur (front-end), le module fonctionnel qui implémente la partie métier de l'application (le back-end) et le module de sauvegarde qui constitue la mémoire de l'éditeur ; c'est elle qui sauvegarde et restaure les documents et leurs modèles (grammaires) manipulés dans l'outil. La figure 5 donne un aperçu de son architecture générale : c'est une instance de la méta-architecture présentée sur la figure 1.

Les différents composants de cette architecture sont les suivants :

– *Les interfaces utilisateur de TinyCE :* on dispose d'une interface utilisateur pour les clients (où s'effectue l'édition) et d'une autre pour le serveur (où s'effectue la fusion). L'interface associée au serveur (voir figure 6) permet l'édition de la grammaire (symboles, productions, axiome), des différentes vues, du document (global). Elle permet aussi de préciser les adresses des différents postes clients et de leur expédier une vue de la grammaire et éventuellement un réplicat partiel du document (global) courant. L'interface associée aux clients (voir figures 7) permet la connexion au serveur pour le rapatriement de la grammaire (projetée) ainsi que du réplicat partiel. Elle permet aussi l'édition et la validation de ce réplicat partiel ainsi que sa re-expédition vers le serveur.

– *Le module fonctionnel de TinyCE :* il contient le code (Haskell) des routines permettant de réaliser la *projection*, l'*expansion*, la *validation*, la *fusion*, ainsi qu'un certain nombre de *codes de services*, essentiellement des *transducteurs* utilisés pour le reformatage (codage/décodage) des données provenant de l'interface utilisateur sous forme de document XML.

– *Le module de sauvegarde de TinyCE :* il permet la sérialisation/restauration à la XML des grammaires et des documents .

8. What You See Is What You Get.
9. Le poste serveur est celui sur lequel on effectue divers opérations comme celles de *projection* et de *fusion*.

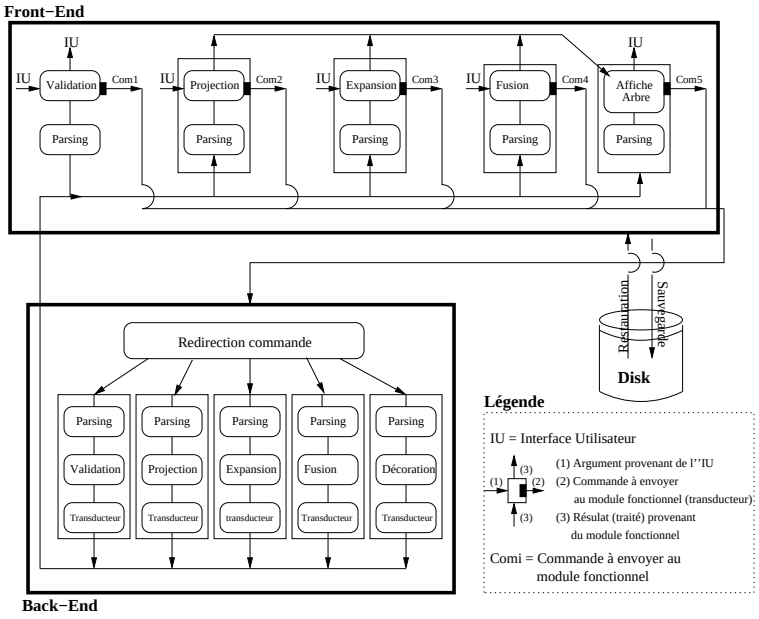

Figure 5. *Architecture de TinyCE*

Cette présentation de la spécification de *TinyCE* montre très clairement qu'il n'est pas un prototype trivial à réaliser car, il possède une IHM complexe (fig. 6 et fig. 7), manipule des données infinies, est déployé sur un réseau, . . .

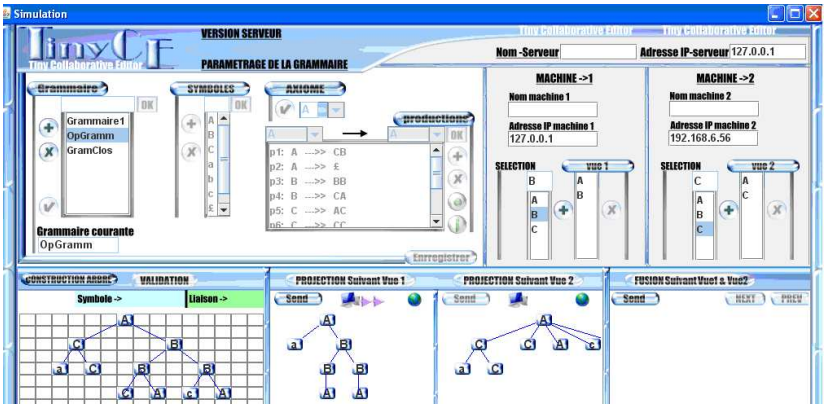

Figure 6. *Interface serveur de TinyCE présentant un début d'édition ainsi que les vues partielles à envoyer aux différents clients.*

3.1.2. Implémentation de TinyCE

L'implémentation de *TinyCE* s'est faite très aisément en suivant notre protocole de programmation : on a utilisé Java pour réaliser l'interface graphique et Haskell pour la partie métier. La communication entre les deux modules (fonctionnel et interface utilisateur) se fait par envoi de messages, typiquement, un document XML suivant le protocole

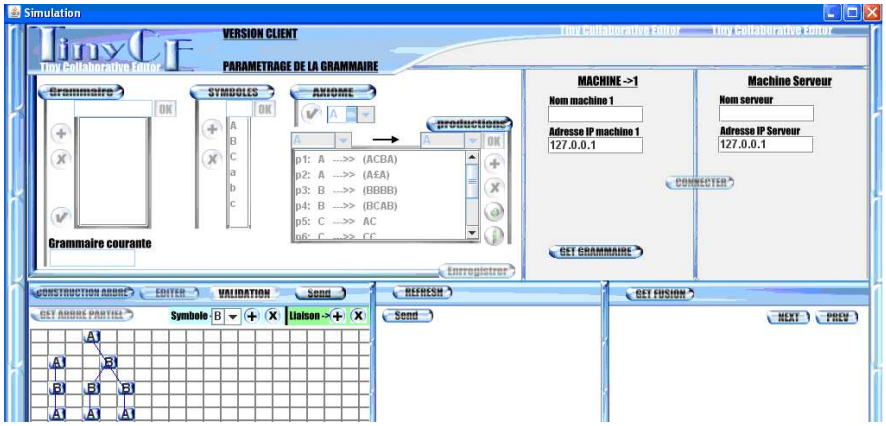

Figure 7. *Interface du premier client de TinyCE présentant la mise à jour locale de la vue partielle à envoyer au serveur.*

présenté dans les sections précédentes. A titre d'illustration, nous présentons ci-dessous un exemple de réalisation d'une opération - *la projection* - dans *TinyCE*.

Illustration : Pour la réalisation de l'opération de projection effectuée sur un document, après action (clic) sur le bouton *"Projection Suivant vue i"* depuis l'interface graphique, la routine Java correspondante du *moteur* qui "écoute" ce bouton exécute une instruction de la forme *"ExecuteExternProgramInJava.StartCommand(com)"* dans laquelle, comme dans le code de la page 223, *com* est un tableau à deux entrées tel que : *com[0]* contient *"moduleMetier.exe"*, le chemin d'accès (relatif) au fichier exécutable (Haskell) contenant le code métier de l'application. *com[1]* contient *"<commande> . . . </commande>"* (voir fichier complet ci-dessous) : un document XML valide encapsulant le nom de l'opération à effectuer ainsi que ses arguments. Par exemple, la projection suivant la vue *{A, B}* du document *t= Node A [Node C [Node a [], Node C []], Node B [Node B [Node c [], Node A []], Node B [Node A [] Node A]]]*, pourra correspondre au document XML suivant :

```
<Commande>
   <NomCommande fichier = "moduleMetier.exe"> projection </NomCommande>
       <Arguments>
          <Arg type = "String"> Node A [Node C [Node a [], Node C []],
              Node B [Node B [Node c [], Node A []], Node B [Node A [] Node A]]]
          </Arg>    <!-- Le document "t" -->
          <Arg type = "String"> A, B </Arg>   <!-- La vue -->
          <Arg type = "String"> #linearisation grammaire# </Arg>
       </Arguments>
</Commande>
```

A la réception de ce document dans le module fonctionnel, le transducteur l'analyse pour en extraire les informations utiles (arbre à projeter (*t*), vue, grammaire), les met sous les types adéquats puis, les transmet comme paramètres d'appel de la fonction *projection*. A la fin de l'exécution de cette fonction, un document XML contenant le résultat de l'exécution est construit et transmis au moteur pour traitement. Ce document est semblable à celui qui suit :

```
<Resultat reponseStructure = "No" type Element = "String">
   Node A [Node a [], Node B [Node B [Node A []], Node B [Node A [] ]]]
</Resultat>
```

4. Conclusion

Dans ce papier, nous avons présenté un protocole de programmation dit de fertilisation croisée entre un langage fonctionnel (Haskell) et un langage objet (Java), une méta-architecture pour les applications à développer suivant ce protocole, ainsi qu'une instanciation de celle-ci pour la réalisation de *TinyCE* qui nous a permis de valider notre approche de la fertilisation croisée. Le développement des applications suivant ce protocole procure de nombreux avantages : manipulation aisée des données infinies, prototypage rapide des applications, modularité, concision ....

D'un certain point de vue, en matière de fertilisation croisée, la vraie difficulté de la communication entre Haskell et Java consiste à savoir comment passer aux programmes Java des structures de données complexes, des types abstraits, des fonctions... ? Inversement, comment passer des objets, et des classes aux programmes Haskell ? Ce point de vue est très intéressant et pourra faire l'objet d'un travail futur. Toutefois, dans le présent travail, nous avons abordé la problématique de la fertilisation croisée Haskell-Java suivant une autre perspective. Dans notre approche, nous souhaitions une solution simple à comprendre, facile à mettre en oeuvre, d'appropriation facile, et donc reproductible pour d'autres couples de langages[10], nous avons plutôt proposé un protocole inspiré de celui des RPCs. Dans notre approche, au moment de l'invocation d'une routine écrite dans l'autre langage, il suffit de considérer un ensemble de correspondance entre les types de base des différents langages puis, de laisser le soin au transducteur de reformater dans le type approprié les données utilisées pour l'invocation. On procède de même pour le résultat.

Plusieurs critiques peuvent être portées à ce travail. Entre autre, vu que nous avons opté pour une solution simple, on peut lui reprocher d'être peu robuste. En effet, on n'a aucune garantie statique que les messages seront bien interprétés à leur réception (pas de vérification de typage). On peut envisager au moins deux solutions à cette limitation : la première consiste à inclure dans le message des informations qui seront utilisées par le transducteur lors du décodage ou de l'analyse (parsing) du message pour effectuer des vérifications. Il s'agit donc tout simplement de changer le format des messages et d'ajouter quelques instructions dans le transducteur. La seconde solution est inspirée de ce qui se fait dans les web services à savoir, publier pour chaque module (fonctionnel ou IHM) les services offerts ainsi que leurs profils.

Une fertilisation croisée entre Haskell et plusieurs autres langages est également étudiée dans [13] : *The Haskell 98 Foreign Fonction Interface (FFI)*[11]. C'est un travail de bien plus grande envergure que celui présenté ici. Toutefois, notre approche a l'avan-

10. Rappelons que le couple Haskell-Java utilisé dans ce papier n'est qu'illustratif par rapport au protocole. La solution proposée est reproductible à d'autre couples de langages. En effet, les concepts techniques et même technologiques que nous avons utilisés ne sont pas inhérents aux seuls langages Haskell et Java, ni même aux seuls paradigmes fonctionnel et objet.

11. C'est à notre connaissance le seul travail important effectué en rapport avec la problématique traitée dans ce papier.

tage d'être plutôt naturelle et donc d'appropriation facile par opposition à celle présentée dans [13] où, l'utiliser nécessite au moins l'apprentissage d'une nouvelle syntaxe, ce qui constitue un effort intellectuel en plus à effectuer qui peut s'avérer être un effort en trop. Bien plus, il faut aussi acquérir et installer une implémentation correcte des *FFI* prenant en charge les langages qu'on souhaite utiliser.

Les méthodes d'analyse et de conception des systèmes ne sont pas souvent pures : on peut trouver par exemple dans des méthodes objets des spécifications qui sont empruntées à des méthodes fonctionnelles (ou autre) afin de pouvoir mieux appréhender une préoccupation. Si l'analyse et la conception d'un système sont faites en utilisant une approche objet dans laquelle on retrouve des bribes d'analyse et de conception plus ou moins fonctionnelles, le passage de cette phase d'analyse-conception à celle d'implémentation (objets par exemple) nécessitera une traduction des éléments de modélisation fonctionnelle en des éléments de modélisation (ou d'implémentation) objets, ce qui est loin d'être commode et naturel. Un travail plus important pouvant être abordé à la suite de ce papier consiste à étudier plus précisément et de formaliser comment peut s'effectuer la traduction des éléments de modélisation issus d'une conception hybride en des éléments de mise en oeuvre correspondant dans le langage hybride issu de la fertilisation croisée.

5. Bibliographie

[1] W.E. Carlson, P. Hudak, and M.P. Jones. An experiment using Haskell to prototype "geometric region servers" for navy command and control. *Research Report 1031, Department of Computer Science, Yale University*, November 1993.

[2] P. Henderson. Functional programming, formal specification, and rapid prototyping. *IEEE Transactions on SW Engineering*, SE-12(2) :241-250, 1986.

[3] Why Haskell matters, http ://www.haskell.org/haskellwiki/Why_Haskell_matters

[4] J. Fokker. Functional Parsers. In J. Jeuring and E. Meijer, editors, *First International School on Advanced Functional Programming*, volume 925 of Lecture Notes in Computer Science, 1-23, Springer-Verlag, 1995.

[5] CUP : LALR Parser Generator in Java, http ://www2.cs.tum.edu/projects/cup/

[6] Haskell, A Purely Functional Language. http ://www.haskell.org

[7] Java, un langage Orienté Objets. http ://www.java.com

[8] GUI Libriries, http ://www.haskell.org/haskellwiki/Applications_and_libraries/GUI_libraries

[9] XSL. http ://www.w3.org/TR/xsl/.

[10] The Glasgow Haskell Compiler : GHC. http ://haskell.org/ghc/.

[11] Eric Badouel and Maurice Tchoupé T. Merging hierarchicallystructured documents in workflow systems. *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science(CMCS 2008)*, Budapest. Electronic Notes in Theoretical Computer Science, 203(5) :3-24, 2008.

[12] Maurice Tchoupé T. , Une approche grammaticale pour la fusion des réplicats partiels d'un document structuré : application à l'édition coopérative asynchrone. *Thèse de Doctorat/PhD, Université de Rennes I/Université de Yaoundé I, 2009.*

[13] M. Chakravarty,S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. Peyton Jones, A. Reid, M. Wallace, M. Weber, The Haskell 98 Foreign Function Interface 1.0 : An Addendum to the Haskell 98 Report, http ://www.cse.unsw.edu.au/∼chak/haskell/ffi/, 2005

6. Annexe

Dans cette annexe, nous donnons le code de quelques classes et fonctions ayant servi à la mise en oeuvre de l'exemple portant sur le générateur de nombres premiers présenté dans ce papier. Ces fonctions sont données afin qu'un lecteur désirant expérimenter notre approche de la fertilisation croisée Haskell-Java puisse s'en inspirer (surtout s'il n'est pas un habitué de XML) pour écrire par exemple son transducteur (parseur) ou tout autre module de son application. Ce code contient essentiellement les classes suivantes :

– Le fichier *IhmHaskellDansJava.java* donne le code de l'interface utilisateur.

– Le fichier *ExecuteExternProgramInJava.java* est le module permettant d'invoquer une routine Haskell dans Java. Un lecteur pourra le recopier en l'état.

– Le fichier *Transducteur.java* donne le code de la routine *parseur* permettant de transformer un fichier XML conforme à la DTD de la page 221 en un document XML ne contenant que le résultat de l'exécution du programme Java invoqué. Cette transformation est effectuée au moyen d'une feuille de style XSLT.

```
1  //Fichier IhmHaskellDansJava.java
2  import javax.swing.*;
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5
6  class HaskellDansJava implements ActionListener {
7       //Les composants de l'IHM
8     JLabel nbreNbrePremier = new JLabel("Nbre de nbre premier à générer");
9     JTextField nbrePremier = new JTextField(3);
10    JTextArea resultat = new JTextArea( "", 3, 20);
11    JButton generer = new JButton("Generer");
12
13    HaskellDansJava (){
14      JPanel contenuFenêtre = new JPanel();;
15      contenuFenêtre.add(nbreNbrePremier);
16      contenuFenêtre.add(nbrePremier);
17      contenuFenêtre.add(generer);
18      contenuFenêtre.add(resultat);
19                  generer.addActionListener(this);
20                  JFrame cadre = new JFrame("Crible by Haskell-Java");
21      cadre.setContentPane(contenuFenêtre);
22      cadre.setSize(400,150);
23      cadre.setVisible(true);
24    }
25
26    public void actionPerformed(ActionEvent événement) {
27       String[] cmd = new String[2];
28       cmd[0] = "crible.exe" ;
29       cmd[1] = Transducteur.codeur("crible.exe", "nPremier", "Integer 10");
30       String resulHaskell = ExecuteExternProgramInJava.StartCommand(cmd);
31         /*chaine de caractères représentant le fichier XSLT*/
32       String xsl = "<xsl:stylesheet version=\"2.0\" " +
33                    "xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">" +
```

```
                    "<xsl:template match=\"Resultat\"> "+
                       "<xsl:value-of select=\".\"/>"+
                    "</xsl:template>"+
                  "</xsl:stylesheet>";
      resulHaskell = Transducteur.parseur(resulHaskell, xsl);
      resultat.setText(resulHaskell);
   }
   public static void main(String[] args) {
      HaskellDansJava monIHM = new HaskellDansJava();
   }
}

//Fichier ExecuteExternProgramInJava.java
class ExecuteExternProgramInJava {
   //passage par argument de la commande à lancer
   static String StartCommand(String [] command) {
     try {
        //creation du processus
        Process p = Runtime.getRuntime().exec(command);
        InputStream in = p.getInputStream();
          //on récupère le flux de sortie du programme
        StringBuffer build = new StringBuffer();
        char c = (char) in.read();
        while (c != (char) -1) {
            build.append(c);
            c = (char) in.read();
        }
        String resultat = build.toString();
        return resultat;
    }
    catch (Exception e) {
       System.out.println("\n" + command + ": commande inconnu ");
       return "";
       }
    }
    }
```

6.1. Exemple de code du parseur (côté Java)

```
//Fichier Transducteur.java
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.Source;
import java.io.StringReader;
import java.io.StringWriter;
import javax.xml.transform.OutputKeys;

class Transducteur  {
```

```
  static String codeur(String nomFichier, String nomFonction, String listeParam){
  ...
  return chCode;
  }
  static String parseur(String inputXml, String inputXsl){
    try {
      Source xmlInput = new StreamSource(new StringReader(inputXml));
      Source xslInput = new StreamSource(new StringReader(inputXsl));
      StringWriter stringWriter = new StringWriter();
      StreamResult xmlOutput = new StreamResult(stringWriter);
      Transformer transformer = TransformerFactory.newInstance().newTransformer(xslInput);
      transformer.transform(xmlInput, xmlOutput);
      return xmlOutput.getWriter().toString();
      } catch (Exception e) {
         throw new RuntimeException(e); // simple exception handling, please review it
        }
   }
 public static void main(String[] args) throws Exception{
   /*chaine de caractères représentant le fichier XML*/
  String xml = new String("<Resultat reponseStructure = \"Yes\" typeElement = \"int\" "+
                           "separateur = \",\"> 2,3,7,11,13,17,19,23,29 </Resultat>");
   /*chaine de caractères représentant le fichier XSLT*/
  String xsl = "<xsl:stylesheet version=\"2.0\" " +
               "xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">" +
               "<xsl:output method=\"text\"  />" +
               "<xsl:template match=\"Resultat\"> "+
                   "<xsl:value-of select=\".\"/>"+
               "</xsl:template>"+
              "</xsl:stylesheet>";
  String result = parseur(xml, xsl);
  System.out.println(result);
  }
}
```