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 basée 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, Édition coopérative, Parseurs, Évaluation presseuse, 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 pour 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 mathématiques (λ calcul) qui permet d'y écrire des programmes dont on peut aisément prouver l'exactitude) ils ont longtemps peiné à être adopté par les programmeurs pour le développement logiciel.

Dans les années 90, Hudak et al. dans [1] tout en faisant le constat selon lequel 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 prouver l'exactitude... Une comparaison des prototypes de la même application réalisés avec Haskell, Ada et C++ respectivement, comparaison portant sur le nombre de ligne de codes, 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 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 part 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. 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 la programmation 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 que la 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 puis, un prototype d'éditeur coopératif mis en oeuvre suivant notre approche. La section 4 est consacrée à la conclusion.

2. Fertilisation croisée Haskell-Java

Cette section présente notre approche de la fertilisation croisé 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].

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 croisé Haskel-Java est schématisé sur la figure 1. Il s'agit ici à partir de deux programmes écrits dans les langages Java et Haskell respectivement d'insérer dans le programme Java d'une ou plusieurs entrées à partir desquelles on appellera un ou plusieurs programmes Haskell en leurs transmettant des valeurs correspondants aux arguments des programmes appelés. Le programme Haskell s'exécute et retourne un résultat qui est *intercepté* dans le programme Java. Ainsi présenté, le seul problème qui subsiste est celui relatif au format dans lequel les paramètres doivent être codés. Nous préconisons pour cela un format à la XML. 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 appelé *transducteur*[3] qui effectue les tâches suivantes :

1) A l'appel du programme Haskell, le transducteur (*codeur*) réalise préalablement un codage 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) conve-

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 incompletes [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. 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* et le *parseur*.

nable 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.

4) A la réception du résultat d'exécution du programme Haskell dans le programme Java, le transducteur (*parseur*) analyse le document résultat qui lui est acheminé et le rend dans le type convenable utilisé dans la suite du programme Haskell. On en déduit (figure 1) une méta-architecture pour les programmes à développer suivant ce protocole.

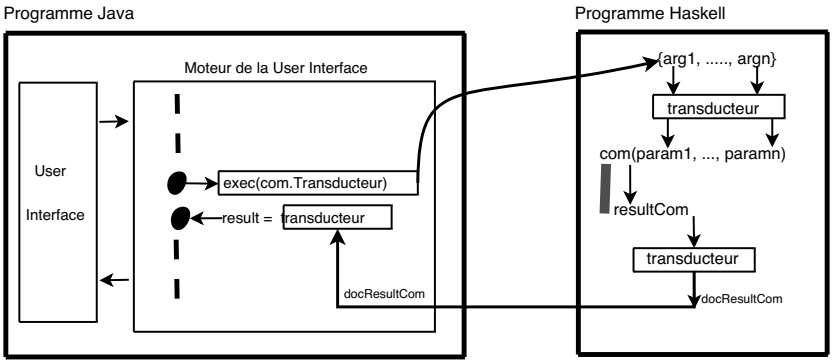

Figure 1. *Méta-architecture des applications à développer par fertilisation croisée Haskel-Java*

L'élaboration 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].

2.2. Mise en oeuvre et illustration

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[4] enregistré par exemple sous le nom *crible.hs*.

```
1  module Main where
2  import System.Environment
3  main = do
4              args <- getArgs
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 etre positif!"
9                    else do take n (crible [2..])
10 crible [] = []
11 crible (n:ns) = n:(crible [m | m <- ns, (m `mod` n) /= 0])
```

4. 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.

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 en ligne de commandes *"crible x"* où *x* est l'argument de l'appel.[5]

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.

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 un exemple d'appel du programme Haskell conçu précédemment dans un programme java[6].

```
1  import java.io.*;
2
3  class HaskellDansJava {
4     //passage par argument de la commande à lancer
5     static String StartCommand(String [] command) {
6       try {
7           //creation du processus
8           Process p = Runtime.getRuntime().exec(command);
9           InputStream in = p.getInputStream();
10            //on récupère le flux de sortie du programme
11          StringBuffer build = new StringBuffer();
12          char c = (char) in.read();
13          while (c != (char) -1) {
14               build.append(c);
15               c = (char) in.read();
16          }
17          String resultat = build.toString();
18          return resultat;
19      }
20      catch (Exception e) {
21         System.out.println("\n" + command + ": commande inconnu ");  return ""; }
22      }
23      public static void main(String [] args){
24         String[] cmd = new String[2];
```

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

6. Notons que de part le fait que le programme Haskell réalisant le crible n'utilise qu'un seul paramètre (le nombre de nombres premiers à générer) qui en plus est de type simple - un entier -, nous n'avons pas (ayant à coeur d'alléger la présentaion) respecté scrupuleusement dans ce code le protocole de programmation ainsi que la méta-architecture présentée à la section précédente ; ce sera toutefois le cas dans l'exemple développé à la section suivante.

```
25          cmd[0] = "crible.exe" ;
26          cmd[1] = "10"; //les 10 premiers nombres premiers
27          System.out.println("Appel du programme Haskell");
28          String resultat = StartCommand(cmd);
29          System.out.print("Resultat de l'execution du programme Haskell: ");
30          System.out.println(resultat);
31          }
32  }
```

L'exécution du programme précédent produit la sortie suivante :

```
1  Appel du programme Haskell
2  Resultat de l'execution du programme Haskell: [2,3,5,7,11,13,17,19,23,29]
```

3. TinyCE : un prototype d'éditeur coopératif asynchrone

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 basée sur la notion de vue partielle (des documents). Une vue partielle ou réplicat partiel d'un document édité de façon coopérative correspond aux différents objets (certaines parties) du document qui sont pertinents pour un utilisateur particulier donné participant à l'édition.

Ci-dessous nous présentons brièvement comment l'implémentation d'un prototype d'éditeur coopératif (nommé *TinyCE* pour "*a Tiny Cooperative Editor*") utilisant cette notion de vue a été menée au moyen d'une fertilisation croisée Haskell-Java. Notons 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 une liste infinie d'éléments.

TinyCE [12] est un éditeur WYSIWYG[7] permettant l'édition conviviale (édition graphique par simple utilisation de la souris) et coopérative de la structure abstraite (arbre de dérivation) des documents structurés. Il s'utilise en réseau suivant un modèle client-serveur. Son interface utilisateur offre à l'usager des facilités pour l'édition du modèle des documents (une grammaire) et 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 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

7. What You See Is What You Get.

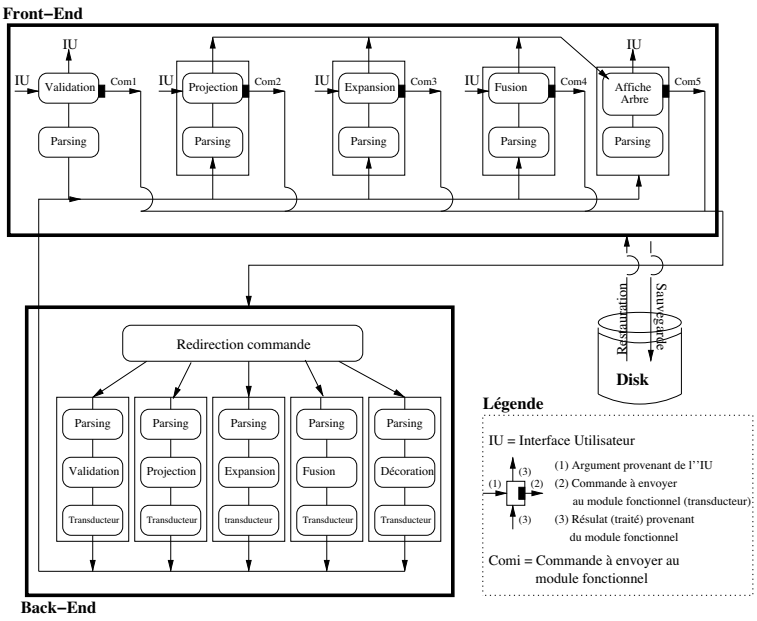

Figure 2. *Architecture de TinyCE*

documents et leurs modèles (grammaires) manipulés dans l'outil. La figure 2 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 une autre pour le serveur (où s'effectue la fusion). L'interface associée au serveur (voir figure 3) 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 permet la connexion au serveur suivie du 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 après avoir précisé l'identité du poste serveur dans le champ approprié de l'interface.

– *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 .

3.1.2. Implémentation de TinyCE

L'implémentation de *TinyCE* est effectuée au moyen de deux langages : Java pour l'interface graphique et Haskell pour la partie métier. La communication entre les deux modules (fonctionnel et interface utilisateur) se fait par envoie de messages, typiquement, un document XML.

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 doit exécuter une instruction de la

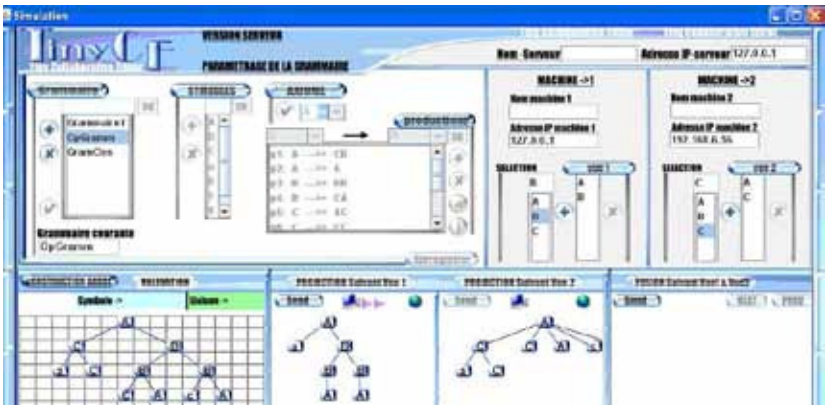

Figure 3. *Interface serveur de TinyCE présentant un début d'édition ainsi que les vues partielles à envoyer aux différents clients.*

forme *"Runtime.getRuntime().exec(com)"* dans laquelle, comme dans le code de la page 4, *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>"* : 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 []] . . .* , pourra correspondre au document XML suivant :

```
<commande>
  <nomOp> projection </nomOp>    <!-- Le nom de l'opération -->
  <arguments>
    <arg> Node A [Node C [Node a [], Node C []],
           Node B [Node B [Node c []], Node B [Node A]]] </arg> <!-- Le document "t" -->
    <arg> A, B </arg>    <!-- La vue -->
    <arg> #linearisation grammaire# </arg>
  </arguments>
</commande>
```

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

4. Conclusion

Dans ce papier, nous avons présenté un protocole de programmation dit de fertilisation croisée entre un langage fonctionnel (Haskell) et un autre 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*. Le développement des applications suivant ce protocole procure de nombreux avantages : manipulation aisée des données infinies, prototypage rapide des applications, modularité, concision . . . .

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

donc d'appropriation facile par opposition à celle présentée dans [13] où, il faut non seulement apprendre une nouvelle syntaxe (appropriation pas facile) mais aussi, acquérir et installer une implémentation correcte des *FFI* prenant en charge les langages que nous souhaitons 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 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 naturelle. Une suite naturelle à ce papier consiste à étudier plus précisément et de formaliser comment s'effectue la traduction des éléments de modélisation issus d'une conception hybride en des éléments de mise en oeuvre correspondants dans le langage hybride issue 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) :241250, 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) :324, 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