Ecole Polytechnique Fédérale de Lausanne

Laboratoire des Méthodes de Programmation

Recherche d'information dans un programme Scala

Tchakouté Roland

Février 2004

Projet de semestre

Assistant responsable : Vincent Cremet
Professeur responsable : Martin Odersky

Table des matières

1. Introduction

Pour écrire un programme en utilisant une bibliothèque, il est toujours nécessaire d'avoir accès à l'interface documentée de cette bibliothèque pour déterminer quelle classe on peut réutiliser, pour se rappeler quelles méthodes on peut appeler sur un objet d'un type donné.

La solution la plus basique consiste bien sûr à inspecter les sources de la bibliothèque, si celles-ci sont disponibles, mais l'interface de la bibliothèque est alors noyée dans le code d'implantation. De plus le programmeur doit rechercher lui-même l'emplacement de la définition correspondant à l'usage d'un nom, par exemple la définition d'une classe apparaissant dans la signature d'une fonction.

La solution la plus répandue consiste alors à présenter une vision statique de la bibliothèque en omettant toute la partie implantation (le corps des fonction et la définition des parties privées) et en fournissant des liens pour accéder immédiatement à la définition d'un nom donné. C'est l'exemple de Javadoc [4]. Mais cette situation n'est pas complètement satisfaisante car elle présente une information de façon figée qui ne convient pas à toutes les situations. Par exemple on peut vouloir trouver tous les opérateurs qui prennent deux entiers et renvoient un entier, ou toutes les fonctions dont le nom répond à un critère donné ou dont le commentaire contient un ensemble de mots clés.

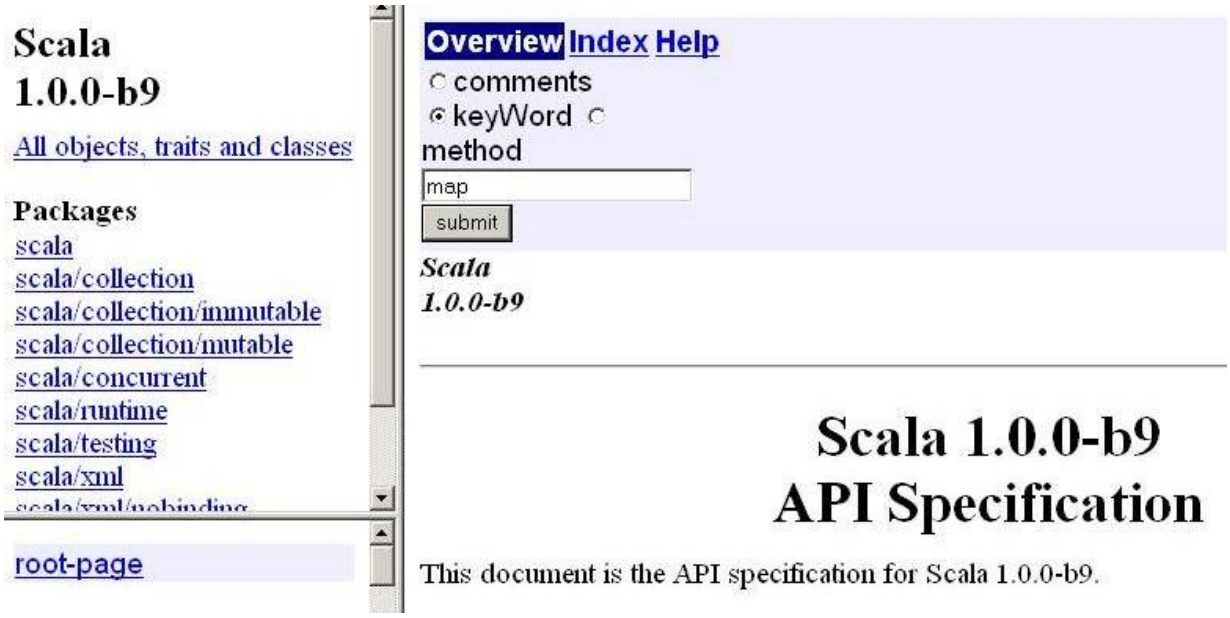
Le but de ce projet est d'ajouter ces fonctionnalités à Scaladoc, l'outil de génération de documentation qui accompagne le langage de programmation Scala. Scala [3] est un nouveau langage de programmation qui combine de manière judicieuse les avantages de la programmation fonctionnelle et ceux de la programmation orientée objet. La recherche par types se fait modulo isomorphisme de types. Ne pas distinguer les types isomorphes permet principalement de se passer des détails peu importants comme par exemple l'ordre des paramètres. Il existe un ensemble d'axiomes qui caractérisent cet isomorphisme entre types.

2. Exemple d'utilisation

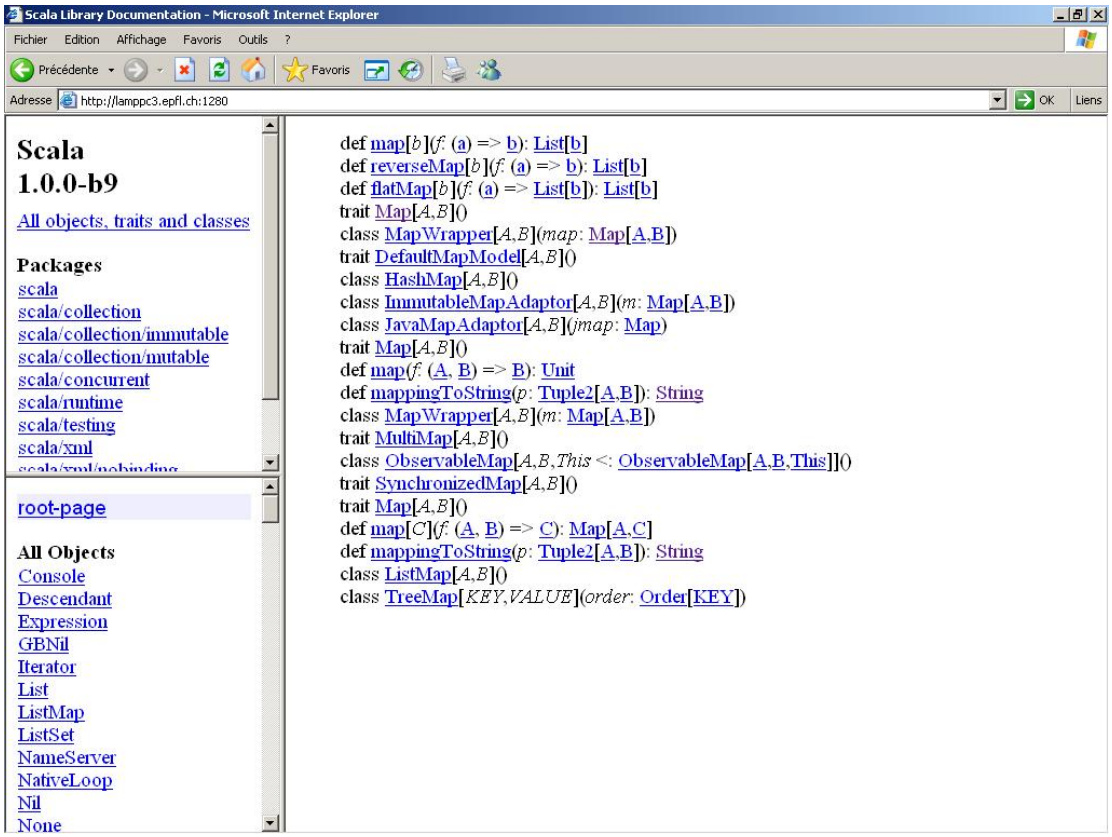
2.1 Côté serveur

L'exécution de la commande scaladoc avec un ensemble de fichiers Scala comme arguments génère les pages HTML statiques pour ces fichiers et lance un serveur HTTP capable d'envoyer ces pages HTML et de répondre à une requête.

2.2 Côté client



La figure précédente montre une vue de l'interface graphique avec le champ de saisie. L'utilisateur est sur le point de faire une requête sur le nom des symboles contenant « map ».



La figure ci-dessus montre le résultat de la recherche correspondant à la requête faite précédemment avec le mot clé « map ».

2.1 coté serveur

3. Etapes du projet

La première partie du projet consiste à écrire un module qui effectue la recherche d'une entité Scala par son nom ou par un mot clé apparaissant dans son nom en utilisant des expressions régulières.

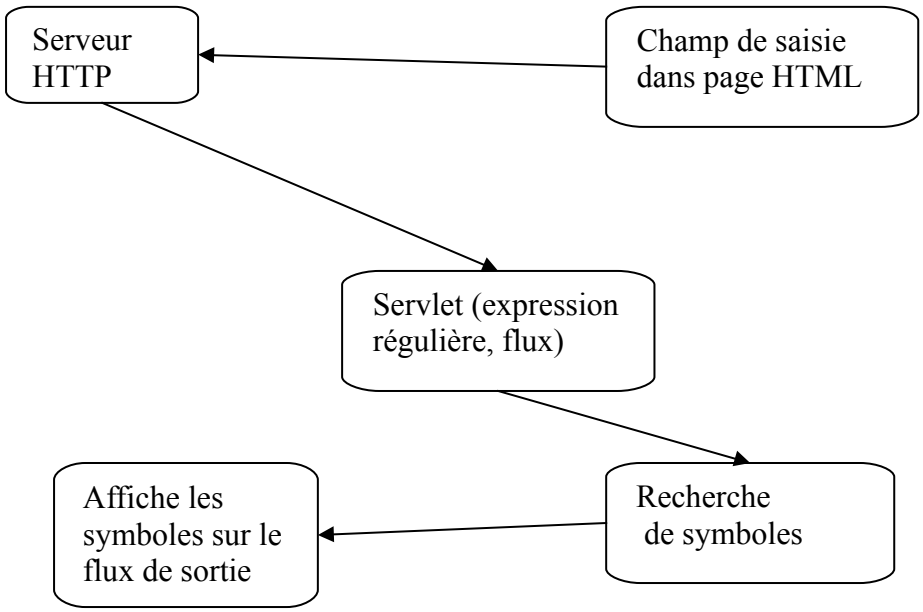
La deuxième partie consiste à écrire un module qui effectue la recherche d'une entité Scala par un mot apparaissant dans ses commentaires en utilisant des expressions régulières. Scaladoc est implanté comme une phase du compilateur qui vient après l'analyse de types. Les deux premières parties ne sont qu'une extension de cette phase. Après la génération normale par Scaladoc des pages statiques associées aux entités Scala de la bibliothèque, une autre page HTML avec la liste des symboles correspondant à la requête (donnée dans un premier temps sous forme d'argument dans le programme) est générée. Les entités apparaissant sur cette page pointent sur les pages statiques générées précédemment.

Dans la troisième phase, on insère un champ de saisie pour les requêtes sur chaque page générée par Scaladoc de façon à pouvoir lancer les requêtes en ligne. Cette phase nécessite l'utilisation d'un serveur HTTP qui puisse accepter une requête, la traiter et y répondre. Dans la version initiale le serveur est lancé automatiquement après la génération des pages statiques.

La quatrième partie du projet consiste à inclure la recherche de méthodes (fonctions) ayant une signature donnée modulo isomorphisme de types.

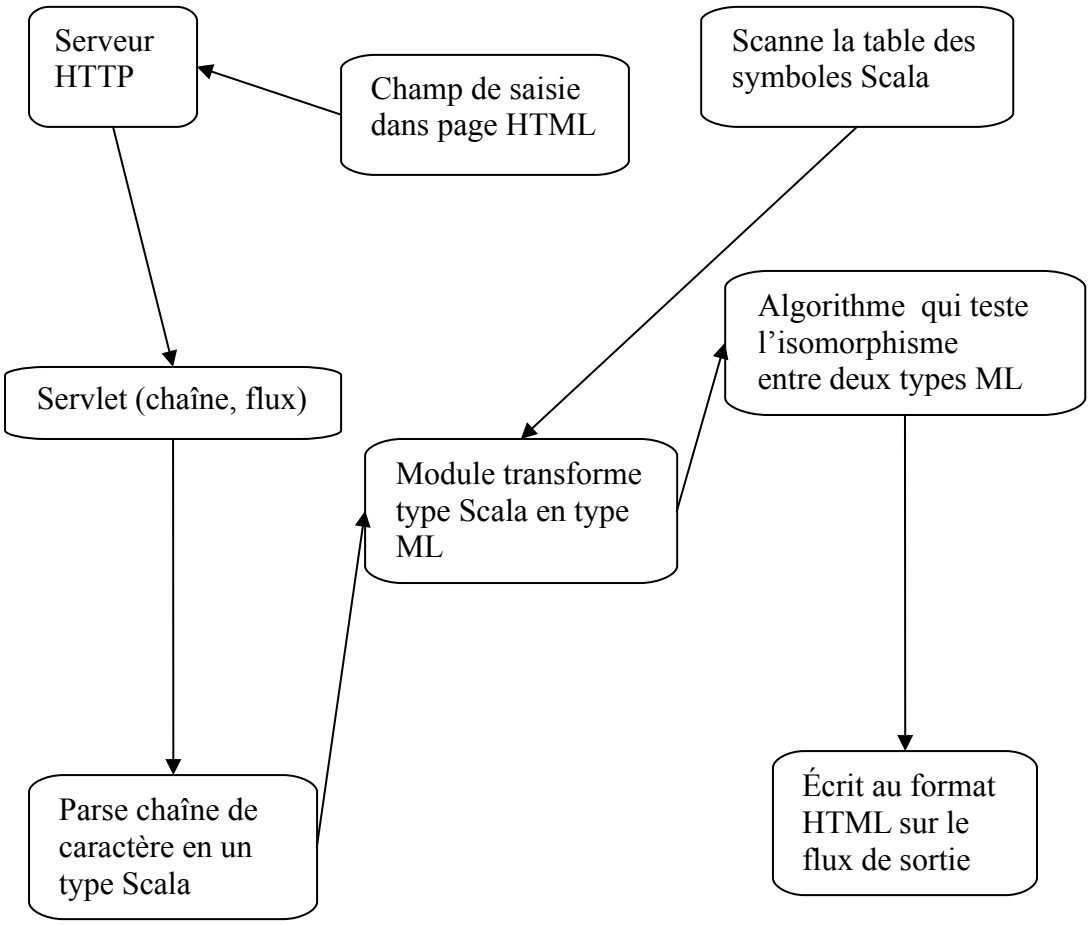
4. Architecture de l'extension de Scaladoc

4.1 Recherche par mots clés



Le champ de saisie est inclus dans toutes les pages HTML générées par Scaladoc. Le serveur HTTP est implanté à partir d'un exemple tiré de la référence[2] de façon à ce qu'il puisse passer une requête à un **servlet** écrit en Java et qu'il ne se contente pas de faire serveur de fichier. Le serveur doit donc être capable de décoder une requête et d'en faire une liste associatives (clé: String, valeur: String).
La recherche de symboles correspond à la fonction qui parcourt la table des symboles et retrouve tous les symboles correspondant à la requête.

4.2 Recherche de fonctions par leur type



On a une fonction qui parse une chaîne de caractères entrée dans le champ de saisie en un type Scala. Le type Scala est ensuite passé en argument à une fonction qui transforme un type Scala en un type ML si c'est possible et qui garde l'association entre les variables de types Scala et les variables de type ML, représentées différemment. D'un autre coté on scanne la table des symbole afin de chercher des fonctions ou des méthodes (vu ici comme des types Scala). Ces types Scala sont ensuite passés en argument à la même fonction qui transforme un type Scala en un type ML. Chacun des types trouvés dans la table des symboles sera passé en argument avec le type parsé plus haut à la fonction qui teste l'isomorphisme entre deux types.

5. Types isomorphes

Définition : Deux types A et B sont **isomorphes** s'il existe deux fonctions de conversion f : A ➔ B et g : B ➔ A tel que g (f(x)) = x quelque soit x de type A, et f (g(y)) = y quel que soit y de type B. Cette définition s'appliquent aux types ML et peut être exprimée sous formes d'axiomes (voir section 5.2).

5.1 Définition des types ML.

Les types ML que l'on considère dans le projet sont définis par induction comme suit.

a (variables de type)

C (constructeurs de type)

A ::= a
¦ C [A_1,….,A_n]
¦ unit

T ::= T ➔ T (type fonctionnel)
¦ T*T (type produit cartésien)
¦A (type simple)

S ::= [a_1,….,a_n] T (schémas de types)

Un schéma de type représente un type polymorphe. On peut par exemple associer à la fonction identité le schéma de type « [a] a ➔ a » Le type « unit » est similaire au « void » qu'on trouve dans les langages impératifs.

5.2 Les axiomes

Comme nous l'avons dit plus haut, la définition mathématique de l'isomorphisme entre types a une présentation équivalente sous forme d'axiomes.

1) A ➔ unit = unit

2) unit ➔ A = A

3) A * unit = A

4) A * B = B * A (commutativité)

5) A * (B * C) = (A * B) * C (associativité)

6) A ➔ (B * C) = (A ➔ B) x (A ➔ C) (distributivité)

7) (A * B) ➔ C = A➔ (B ➔ C) (Currification)

8) X].[Y]A = Y]. X] A

9) X]A = Y]A{Y/X}

10) X]A*B = [X].[Y]A*(B{Y/X})

11) A = unit implique X].A = unit

A, B et C étant des types arbitraires, unit une constante.

L'axiome 7 traduit le fait qu'une fonction qui prend en argument deux valeurs de type x et y (soit un type cartésien x*y) et qui retourne une valeur de type z puisse être considérée comme une fonction qui prend en argument une valeur de type x et qui renvoie une fonction qui prend une valeur de type y en argument et retourne une valeur de type z. On dit alors que la définition de la fonction est currifiée.
L'axiome 10 est souvent appelé « Split ». Il traduit le fait qu'une même variable de type dans deux composantes d'un produit cartésien puisse être considéré comme deux variables de type différentes. Voici un exemple en Scala.

```
def p1[a]: Pair[List[a], List[a]] = Pair(Nil, Nil);

def p2[a,b]: Pair[List[a], List[b]] = Pair(p1._1, p1._2);

def p3[a]: Pair[List[a], List[a]] = p2;
```

Correspondance avec la logique

On peut établir une correspondance entre les types représentés dans les axiomes ci-dessus et les formules logiques du calcul propositionnel ou : (*) est interprétée comme le ET logique, (➔) est interprétée comme l'implication logique, (unit) est interprétée comme la valeur VRAI, X] T) est interprétée comme la quantification universelle sur la variable « X » dans la formule « T ».

6. Traduction des types Scala en types ML

6.1 Rappel sur les types Scala

La représentation interne des types Scala dans le compilateur peut être définis par induction de la manière suivante:

S_cl (symboles de classes)

S_al (symboles d'alias de type)

S_ab (symboles de types abstraits)

S_ty ::= S_cl | S_al | S_ab (symboles de types)

S_te (symboles de termes)

T ::= *ThisType(S_cl)* (type de "this" dans une classe)

| *SingleType(T, S_te)* (types singleton)

| *TypeRef(T, S_ty, {T})* (références de type)

| *CompoundType({T}, <scope>)* (types composés)

| *MethodType({S_te}, T)* (types des méthodes)

| *PolyType({S_ab}, T)* (types polymorphes)

Notation: Un type entre accolade représente une séquence de types.

Les types Scala font référence à des symboles. Par exemple, dans le type *scala.List[int]*, on fait référence au symbole de classe "List" défini dans le package "scala". Il y a trois catégories de symboles de types : les symboles de classe comme dans l'exemple précédent, les symboles d'alias de type comme "T" dans l'exemple *type T = int ;* , et les symboles de types abstraits.
Un type abstrait est soit introduit comme le paramètre de type d'une classe ou d'une fonction, comme dans les exemples *class List[a] { ... }* et *def id[a](x: a): a = x;* , soit c'est un champ de type abstrait dans une classe comme dans l'exemple
type T <: Runnable ; .

Un symbole de terme fait soit référence à un module ("object"), soit à un membre d'une classe ou d'un module introduit par les mots clés "*val*" ou "*de*f", ou soit encore au paramètre d'une fonction ou d'un constructeur.

Revue des différents types Scala

ThisType(S_cl) représente le type de "this" dans la classe "S_cl".

SingleType(T, S_te). "T" est un type qui représente un préfixe, "S_te" est un symbole de terme. Par exemple dans l'exemple suivant :

```
object m1 {
  object m2 {
    val x = 42;
  }
}
```

"m1.m2.x" a le type "m1.m2.x.type", en plus d'avoir le type "int". En fait il n'y qu'un seul objet qui a le type "m1.m2.x.type", c'est "m1.m2.x". C'est pourquoi on appelle ces types des types singletons. "m1.m2.x.type" est représenté en interne par *SingleType(<m1.m2>, <x>)* où "<m1.m2>" est le type associé au préfixe "*m1.m2*" et *"<x>"* est le symbole de "x".

TypeRef(T, S_ty, {T}) représente une sélection de type. Par exemple, le type *scala.Tuple2[int, String]* est représenté en interne par *TypeRef(<scala>, <Tuple2>, [<int>, <String>])*. Il signifie que dans l'objet "*scala*" (qui est ici un package), on sélectionne le type *Tuple2*, qui attend deux arguments de type, et on l'applique aux types *int* et *String*.

CompoundType({T}, <scope>) représente un type composé de plusieurs types et éventuellement raffiné par un ensemble de définitions rassemblées dans une portée

("scope"). Ce type est principalement utilisé en interne pour représenter l'information associée à une classe.

MethodType({S_te}, T) représente le type d'une méthode. Par exemple, dans la définition `def somme(x: int, y: int): int = x + y;` le symbole `somme` a un type représenté en interne par *MethodType([<x>, <y>], <int>)*. Le type des arguments peut-être retrouvé en demandant le type des symboles « *x* » et « *y* ».

PolyType({S_ab}, T) représente le type d'une méthode polymorphe. Par exemple, dans `def first[a, b](p: Pair[a,b]): a;` le symbole *first* aura le type *PolyType([<a>, <b>], <Pair[a,b] => a>)* si <T> est la représentation interne associée au type Scala T.

Exemple

Voici comment est représenté en interne le type *scala.List[java.lang.String]*:

TypeRef(

SingleType(ThisType(<rootclass>), <scala>),

<List>,

[TypeRef(

SinleType(SingleType(ThisType(<rootclass>), <java>), <lang>),

<String>,

[])])

Variables de type

Les variables de type sont représentées par un *TypeRef*. Par exemple, dans le type *[a] scala.List[a] => int* l'occurrence de *a* qui est passé à List est représentée en interne par *TypeRef(Type.localThisType, <a>, Type.EMPTY_ARRAY)*. *Type.localThisType* est défini comme étant *ThisType(Symbol.NONE)*.

Pour tester si un type "*t*" est une variable de type, on peut tester si son symbole associé (*t.symbol()*) est un type abstrait (*isAbstractType())* et si c'est un paramètre (*isParameter()*).

6.2 Définition de la traduction

Voici le schéma de traduction des types Scala en types ML. On note "<<T>>" la traduction du type Scala "T" en schéma de type ML, et "<T>" la traduction d'un type Scala "T" en un type ML.

Définition de <<T>>

<<[a_1,...,a_n] T>>	=	[a_1,...,a_n] <T>
<<T>>	=	<T> si T n'est pas un *polyType*

Définition de <T>

TypeRefs:

<scala.Unit>	=	unit
<scala.Tuple_n[T_1,...,T_n]>	=	<T_1> * ... * <T_n>
<scala.Function_n[T_1,...,T_n, T]>	=	<T_1> -> ... -> <T_n> -> <T>
<localthisType.a[]>	=	a
<p.C[T_1,...,T_n]>	=	C[<T_1>,...,<T_n>]

MethodTypes:

<(T_1,...,T_n) => T>	=	<T_1> -> ... -> <T_n> -> <T>
<() => T>	=	unit -> <T>

Remarquons que la traduction n'est pas définie pour les types singletons, les types de *this* et les types composés.

7. Algorithme de décision de l'isomorphisme entre deux types

La partie sur les isomorphismes a été divisée en deux sous- parties.
J'ai d'abord implémenté un algorithme de décision des isomorphismes entre types ML. Je me suis servi pour cela d'une version écrite en Caml décrite dans le livre « Isomorphisms of types » écrit par Robert Di Cosmo[1]. Cette partie du projet a été écrite en Scala. Dans un premier temps on définit un type algébrique MLType afin de s'abstraire des types et symboles Scala.
Ensuite l'algorithme peut être appliqué à des symboles de fonctions SCALA qui correspondent aux types ML.

7.1. Forme normale d'un type

Chaque type est transformé en sa forme normale qui correspond au type obtenu après les simplifications équivalentes aux axiomes 1, 2, et 3, 5, 6, 7 et 11. Pour cela on définit une fonction de normalisation qui prend en argument un type ML et qui retourne un type ML ne contenant plus de type « unit ». Cette fonction de normalisation peut être décrite sous la forme de règles de réécriture. Pour obtenir ces

règles, on a simplement « orienté« certains axiomes de façon à avoir à la fin un produit de terme [X1]…[Xn](A1*…….*An) ou les termes Ai ne possède plus ni de type unit ni de type cartésien.

7.2 Aplatissage des types et renommage

Ensuite les types normalisés sont aplatis en une paire constituée d'un entier et d'une liste de coordonnées. La longueur de la liste est stockée dans la première composante. Chaque coordonnée (qui n'est rien d'autre qu'un Ai de l'étape précédente) ne contient plus de type cartésien ni de type « unit ».

On effectue ensuite un renommage systématique de chaque variable apparaissant dans les coordonnées de manière à ce qu'une même variable (représentée par un entier) n'apparaisse pas dans deux coordonnées différentes. Cela correspond à l'axiome 10.

Une variable globale est utilisée pour stocker le renommage courant. Au début, le renommage est vide, mais il s'enrichit au fur et à mesure qu'on renomme les variables pour unifier les types dans l'étape suivante.

7.3 Une nouvelle structure de donnée pour les types fonctionnels

Les types fonctionnels sont représentés par une paire « Flat_type », la première composante est elle aussi une paire qui renferme le nombre d'arguments du type fonctionnel ainsi que le type du résultat. La deuxième composante est une liste qui représente les types arguments.

Par exemple le type fonctionnel (int ➔ int) est transformé en sa représentation sous forme de « Flat_type » qui est *Pair (Pair (1,int), List(Pair(0,int),List())))*.

On essaye ensuite d'égaliser deux types fonctionnels aplatis en renommant leurs variables, modulo commutativité et associativité des arguments.

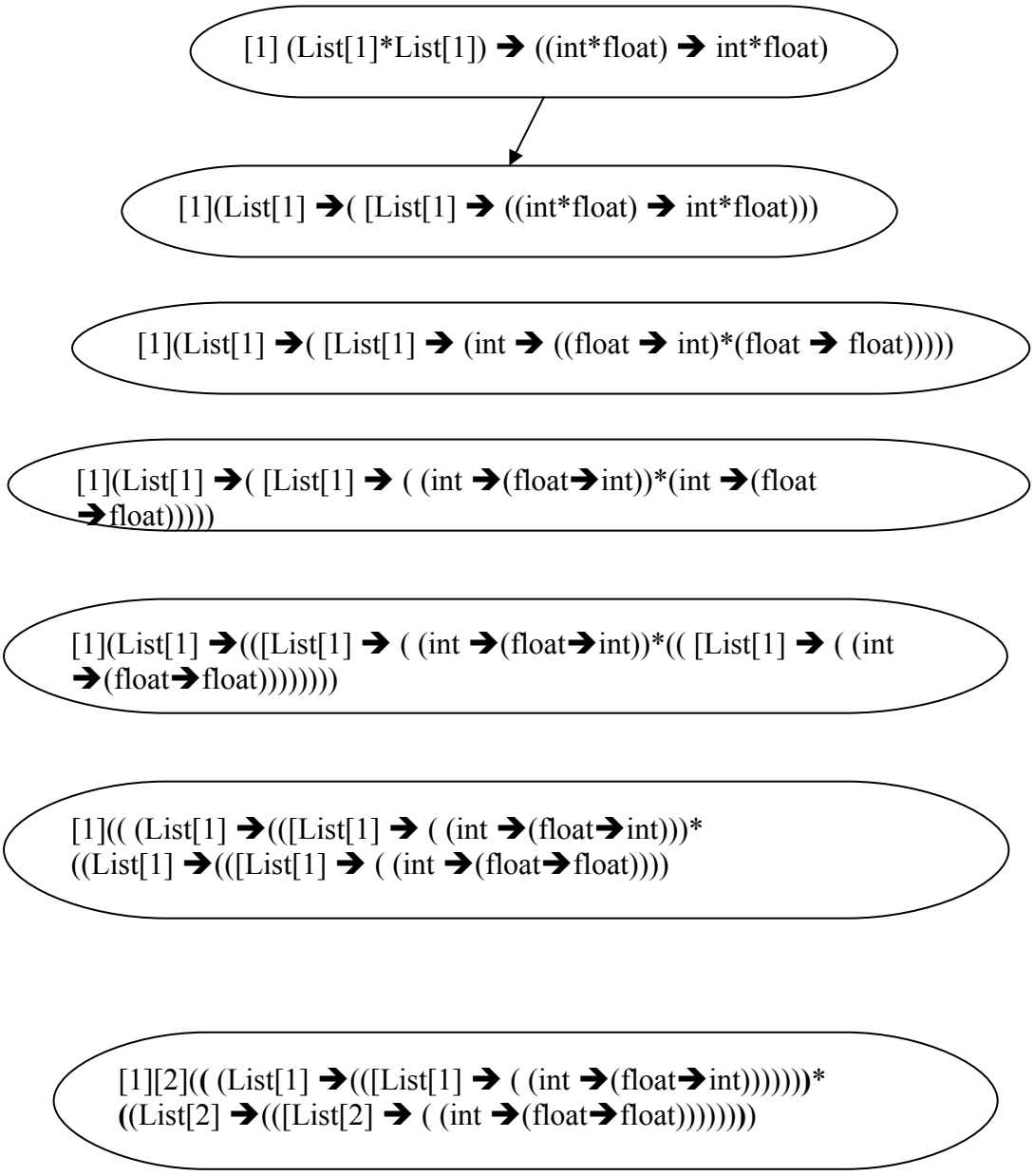
Si c'est possible, la variable global correspondra au renommage qu'il a été nécessaire de construire.

La fonction s'assure d'abord que les nombres d'arguments sont les mêmes. Ensuite si elle parvient à unifier les types de retour, elle essayera d'unifier les types des arguments.

Pour unifier deux listes représentant les arguments des types fonctionnels on génère toutes les permutations de la première liste (l'ordre dans les arguments d'une fonction n'est pas important), et pour chaque permutation on tente de l'unifier avec la deuxième liste.

On sait donc maintenant comment unifier deux coordonnées (représentées par des Flat_types). Il faut maintenant unifier deux *listes* de coordonnées. On utilise pour cela une fonction *findiso* qui prend en argument une liste d'éléments de type « Flat_type » (cette liste est une permutation d'une des deux listes de coordonnées) avec un élément de type « Flat_type » (qui correspond aussi à un type fonctionnel). La fonction tente d'unifier successivement ce dernier avec chaque élément de la liste et s'arrête dès qu'elle y arrive.

7.4 Un exemple



Le schéma ci-dessus est un exemple de normalisation d'un type suivi d'un renommage. On applique à ce type successivement les axiomes 7, 7, 6, 6, 6, 6, et 10.

8. Continuations possibles du projet

Le projet peut être amélioré en plusieurs points. On peut dans un premier temps étendre l'algorithme de décision de l'isomorphisme entre types à l'ensemble des types Scala (ne plus se limiter aux types ML). On devrait alors tenir compte des méthodes définies à l'intérieur des classes et leur ajouter un argument du type de la classe afin qu'elles puisse être vues comme des fonctions normales. On peut ensuite interfacer le module avec un éditeur de texte comme emacs. Le but étant que le programmeur puisse lancer ces modules depuis emacs, et qu'il puisse de manière interactive compléter l'appel à une méthode.

9. Conclusion

Malgré les limitations constatées plus tôt, les nouvelles fonctionnalités apportées à ce projet constitue tout de même un petit pas en avant vers l'amélioration de Scaladoc. Ceci permettra à de nombreux utilisateurs de profiter des autres facilités qu'offre le langage Scala et qui ne sont pas toujours facilement accessibles. J'ai pu mettre en évidence certains concepts appris dans les cours de programmation et de génie logiciel, tel que l'interopérabilité, la modularité, la réutilisation des programmes, les motifs de conception ou design patterns (conteneurs, itérateurs, singleton). D'autre part, la réalisation de ce projet m'a permis de me familiariser avec la programmation fonctionnelle.

10. Annexe

10.1 Définition des types ML en scala

```
class ML {
  /** represents an ML type
  */

  trait TypeConstructor[T <: TypeConstructor[T]] {
    def sameAs(that: T): boolean;
  }
  type TC <: TypeConstructor[TC];

  ...

abstract class MLType;

case object UnitType extends MLType;
case class PolyType(a:TC,L:List[MLType]) extends MLType;
case class TypeVar(n:int)extends MLType;
case class CartesianType(l:MLType,r:MLType) extends MLType;
case class FunctionType (arg:MLType,res:MLType) extends MLType;


...

object ml extends ML {

  class ScalaTC(sym: scalac.symtab.Symbol) extends
TypeConstructor[ScalaTC] {
    val symbol = sym;
    def sameAs(that: ScalaTC): boolean = (sym == that.sym);
  }

  type TC = ScalaTC;
}
```

11. Bibliographie

[1] ROBERTO DI COSMO
Isomorphisms of Types: from λ-calculus to information retrieval and language design

[2] Elliotte Tusty Harold
Programmation réseau avec JAVA

[3] Page web de Scala http://lampwww.epfl.ch/scala/

[4] Javadoc http://java.sun.com/j2se/javadoc/