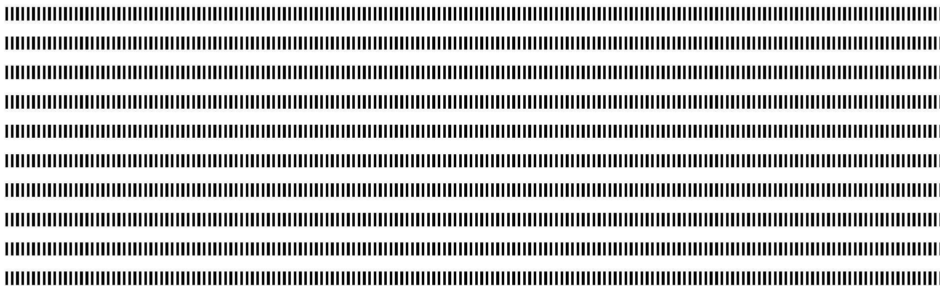

Cohérence de vues dans les grammaires algébriques

Eric Badouel et Maurice Tchoupé

Inria Rennes - Irisa
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
{ebadouel,mtchoupe}@irisa.fr

Maurice Tchoupé bénéficie d'une bourse de thèse offerte par le service de coopération et d'action culturelle de l'ambassade de France au Cameroun et d'un complément financier apporté par le projet SARIMA

RÉSUMÉ. Un document structuré complexe est représenté intentionnellement sous la forme d'une structure arborescente décorée par attributs. Les structures licites sont caractérisées par une grammaire algébrique abstraite. Nous faisons ici abstraction des attributs ; ces derniers sont liés à des aspects sémantiques qui peuvent être traités séparémment des aspects purement structurels qui nous intéressent ici. Cette représentation intentionnelle peut être manipulée de façon indépendante et éventuellement non synchronisée par divers outils d'édition et de manipulation qui opèrent sur des vues partielles distinctes du même document. Pour la resynchronisation de ces vues partielles nous devons résoudre le problème de leur cohérence : décider s'il existe un document correspondant à ces différentes vues et dans l'affirmative produire un tel document. Nous montrons comment résoudre ce problème dans le cas où chaque vue est associée à un sous ensemble des symboles grammaticaux : ceux qui correspondent aux catégories syntaxiques visibles. L'algorithme proposé, qui repose fortement sur le mécanisme d'évaluation paresseuse, résout ce problème même dans le cas où chaque vue partielle peut correspondre à un nombre infini de documents possibles; à la condition toutefois que la donnée conjointe des différentes vues permet de caractériser le document ou, tout au moins, réduit l'ensemble des solutions possibles à un nombre fini de documents.

ABSTRACT. A complex structured document is intentionnally represented as a tree decorated with attributes. The set of legal structures are given by an abstract context-free grammar. We forget about the attributes, they are related with semantical issues that can be treated independently of the purely structural aspects that we address in this paper. That intentional representation may be asynchronously manipulated by a set of independent tools each of which operates on a disctinct partial view of the whole structure. In order to synchronize these various partial views, we are face to the problem of their coherence: can we decide whether there exists some global structure corresponding to a given set of partial views and in the affirmative, can we produce such a global structure ? We solve this problem in the case where a view is given by a subset of grammatical symbols, those associated with the so-called visible syntactical categories. The proposed algorithm, that strongly relies on the mechanism of lazy evaluation, produces an answer to this problem even if partial views may correspond to an infinite set of related global structures; assuming however that the join data of the different views reduces the solution set to a finite set of global structures.

MOTS-CLÉS : DTD, XML, grammaire algébrique, représentation intentionnelle, cohérence de vues, évaluation paresseuse.

KEYWORDS : DTD, XML, contex-free grammar, intentional representation, coherence of views, lazy evaluation.

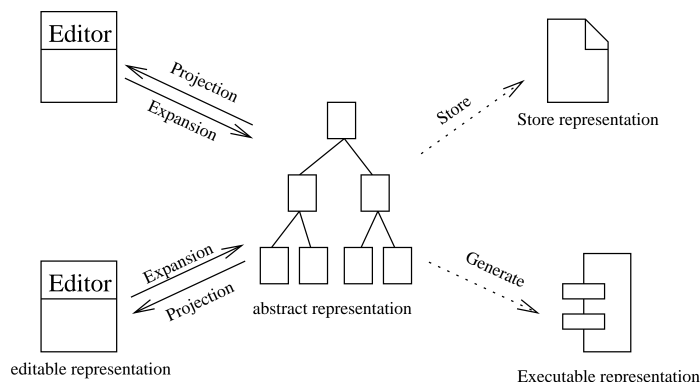

Figure 1. *Un « language workbench » selon[7]*

1. Introduction

Depuis l'avènement de la technologie XML il est usuel d'interfacer des applications par des documents dont la structure doit être conforme à une DTD. Cette technique est par exemple largement utilisée dans l'outil SmartTools [2]. La syntaxe XML est très contrainte et permet de structurer ces documents sous la forme d'arbres de syntaxe abstraite (les construteurs associés aux productions de la DTD sont représentés par une paire de tags dans la syntaxe XML). On peut donc alternativement remplacer ces DTD par des grammaires algébriques abstraites c'est à dire ne contenant pas de symboles terminaux : on fait abstraction de la syntaxe concrète. L'avantage est de pouvoir importer du formalisme des grammaires attribuées les définitions sémantiques qui donnent les dépendances fonctionnelles entre attributs, ce qui n'est pas pris en compte par les DTD.

Depuis leur introduction par Knuth [10] les grammaires attribuées ont été largement utilisées pour l'implémentation des languages de programmation [11]. Dans leur définition d'origine elles combinent à la fois la syntaxe concrète du language, donné par une grammaire algébrique, et les règles sémantiques permettant de définir la valeur des attributs attachés aux différentes catégories syntaxiques. On peut néanmoins séparer l'interprétation (évaluation des attributs) de l'analyse syntaxique en définissant directement les règles sémantiques sur les arbres de syntaxe abstraite. Les règles sémantiques définissent une algèbre pour la signature constituée des constructeurs des arbres de syntaxe abstraite (productions de la grammaire) [5]. Cette algèbre peut être passée en argument à un analyseur syntaxique qui est alors en mesure d'analyser et interpréter le code source sans avoir besoin de produire l'arbre de syntaxe abstraite comme résultat intermédiaire. Cette séparation, qui n'introduit de ce fait aucun coût supplémentaire, procure un certain nombre d'avantages outre la modularité ainsi obtenue. En particulier des travaux récents sur la programmation orientée languages et la programmation générative [12, 4, 7, 3] militent en faveur d'une représentation intentionelle d'un programme, découplée de sa (ou de ses) vue(s) concrète(s) plus ou moins partielle(s), sur laquelle on peut opérer par divers outils de métaprogrammation pour l'éditer, la parcourir, la transformer, en extraire de l'information ...

Nous utiliserons des représentions intentionnelles (pour des programmes ou des données) sous la forme d'arbres de syntaxe abstraite décorés par attributs. Les différents outils d'édition et de manipulation de cette structure opèrent sur des vues partielles de celle-ci.

Par exemple plusieurs intervenants dans un projet, chacun ayant son domaine d'expertise propre, peuvent travailler par le biais d'interfaces qui reposent sur des langages spécifiques liés à leurs domaines d'intervention. On peut imaginer des situations dans lesquelles ces différentes manipulations peuvent se faire de manière asynchrone. Dans un tel cas nous sommes confrontés au problème de la cohérence des vues : existe t'il toujours une structure globale correspondant aux différentes vues à un moment donné ?

2. Grammaires algébriques abstraites et vues

Par *grammaire* nous entendons donc ici la donnée $\mathbb{G} = (\mathcal{S}, \mathcal{P}, A)$ d'un ensemble fini S de *symboles grammaticaux* qui correspondent aux différentes *catégories syntaxiques* en jeu, d'un symbole grammatical $A \in \mathcal{S}$ particulier, appelé axiome, et d'un ensemble fini $\mathcal{P} \subseteq \mathcal{S} \times \mathcal{S}^*$de *productions*. Une production $P = \left(X_{P(0)}, X_{P(1)} \cdots X_{P(n)}\right)$ est notée $P : X_{P(0)} \rightarrow X_{P(1)} \cdots X_{P(n)}$ et $|P| = n$ désigne la longueur de la partie droite de P. Nous représentons en Haskell une grammaire par le type suivant

$$Gram\ \alpha\ \beta\ =\ \beta \rightarrow (\alpha, [\beta])$$

dans lequel les variables de type α et β représentent respectivement les productions et les symboles grammaticaux de la grammaire. Elle est ainsi représentée comme une application qui associe à chaque symbole grammatical X la liste des paires constituée d'une production ayant X en partie gauche et de la partie droite correspondante de cette production. Par exemple la grammaire constituée des productions

$$\begin{array}{lll} P_1 :\ A \rightarrow C\ B & P_3 :\ B \rightarrow C\ A & P_5 :\ C \rightarrow A\ C \\ P_2 :\ A \rightarrow (\,) & P_4 :\ B \rightarrow B\ B & P_6 :\ C \rightarrow C\ C \\ & & P_7 :\ C \rightarrow (\,) \end{array}$$

est définie comme suit

```
data Prod = P1 | P2 | P3 | P4 | P5 | P6 | P7 deriving (Eq,Show)
data Symb = A | B | C  deriving (Eq,Show)
gram : : Gram Prod Symb
gram symb = case symb of
          A -> [(P1,[C,B]),(P2,[])]
          B -> [(P3,[C,A]),(P4,[B,B])]
          C -> [(P5,[A,C]),(P6,[C,C]),(P7,[])]
```

Une vue est un sous ensemble de symboles grammaticaux $\mathcal{V} \subseteq \mathcal{S}$ intuitivement il s'agit des symboles associés aux catégories syntaxiques visibles dans la représentation considérée. On implémentera une vue comme un prédicat sur les symboles :

```
viewAB : : Symb -> Bool             viewAC : : Symb -> Bool
viewAB symb = case symb of            viewAC symb = case symb of
          A -> True                        A -> True
          B -> True                        B -> False
          C -> False                       C -> True
```

A chaque vue est associée une projection sur les arbres de dérivation qui efface les noeuds étiquetés par des symboles non visibles tout en conservant la structure de sous-arbre. Le résultat est une liste d'arbres qui pourra effectivement contenir plusieurs éléments dans le cas où l'axiome est non visible.

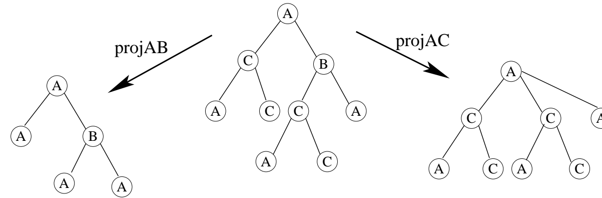

Figure 2. *un arbre de dérivation et ses projections sur {A,B} et {A,C}*

```
data Tree a = Node{top : :a, suc : :[Tree a]}

unode : : Tree a -> (a,[Tree a])
unode (Node x ts) = (x,ts)

projection : : (Symb -> Bool) -> Tree Symb -> [Tree Symb]
projection view der = if view (top der) then [Node (top der) sons] else sons
  where sons = concatenate (map (proj view)(suc der))

concatenate : : [[a]] -> [a]
concatenate xss = [x| xs <- xss, x <- xs]

projAB = projection viewAB       projAC = projection viewAC
```

La figure 2 montre un arbre de dérivation der pour la grammaire et ses deux projections $derAB = projAB\ der$ et $derAC = projAC\ der$.

Il est facile de voir qu'il existe un nombre infini d'arbres de dérivation der correspondant à l'une ou l'autre de ses projections c'est à dire que les ensembles $\{der \mid projAB\ der$ $= derAB\}$ et $\{der \mid projAB\ der\ =\ derAB\}$ sont tous les deux infinis. Par contre un arbre de dérivation pour cette grammaire est caractérisé par l'ensemble de ses deux projections. Nous dirons qu'un ensemble de vues *couvre* une grammaire si tout arbre de dérivation pour cette grammaire est caractérisé par l'ensemble de ses projections. Nous n'avons pour l'instant pas de réponse à la question naturelle suivante :

Problème : *Peut on décider si un ensemble donné de vues couvre une grammaire donnée ?*

3. Une structure de donnée paresseuse pour des ensembles d'arbres de dérivation

Nous introduisons une structure de donnée paresseuse afin de pouvoir représenter et manipuler des ensembles infinis d'arbres de dérivation. Observons qu'une grammaire $gram :: Gram\ Prod\ Symb$, c'est à dire une application $gram :: Symb \rightarrow [(Prod, [Symb])]$, est une co-algèbre pour le foncteur $F\ Prod$ où F est le (bi-)foncteur défini par $F\ \alpha\ \beta =$ $[(\alpha, [\beta])]$. Le point fixe du foncteur $F\ \alpha$ est la structure de donnée paramétrée (par α) suivante

```
data Trees a = Branch [(a,[Trees a])]

unbranch : : Trees a -> [(a,[Trees a])]
unbranch (Branch list) = list
```

Un élément de $Trees\ \alpha$ est une représentation d'un ensemble d'arbres :

```
enumerate : : Trees a -> [Tree a]
enumerate t = [Node elet list_tree | (elet,list_trees) <- unbranch t,
                            list_tree <- dist (map enumerate list_trees)]
```

```
dist [] = [[]]
dist (xs :xss) = [y :ys | y <- xs, ys <- dist xss]
```

L'anamorphisme associé à une co-algèbre permet d'interpréter chaque élément du domaine de la co-algèbre comme un germe (ou un générateur) à partir duquel on peut engendrer un élément de $Trees\ \alpha$.

```
type Coalg a b = b -> [(a,[b])]

ana : : Coalg a b -> b -> Trees a
ana coalg gen = Branch [(a, map (ana coalg) gens)| (a,gens)<- coalg gen]
```

On remarque que si $gram :: Gram\ Prod\ Symb$ est une grammaire, vue comme coalgèbre, et $Axiom$ est l'axiome de la grammaire, alors $ana\ gram\ axiom ::\ Trees\ Prod$ procure une représentation de l'ensemble des arbres de dérivations de la grammaire. Par ailleurs on peut définir un opérateur binaire $\langle\$\rangle$ sur les $Trees\ \alpha$ tel que $T_1\ \langle\$\rangle\ T_2$ représente exactement la liste des éléments dans l'intersection des ensembles d'arbres représentés respectivement par T_1 et T_2.

```
infix 4 <$>

(<$>) : : (Eq a) => Trees a -> Trees a -> Trees a
t1 <$> t2 = Branch (filter void
                   [(a1,zipWith (<$>) ts1 ts2)|
                                   (a1,ts1)<-unbranch t1,
                                   (a2,ts2)<-unbranch t2,
                                    a1==a2,
                                   (length ts1)==(length ts2)])
  where void (a,ts) = not (or (map (null.unbranch) ts))
```

La fonction qui teste l'appartenance d'un arbre $tree :: Tree\ \alpha$ à un élément de $trees ::\ Trees\ \alpha$ peut alors être définie comme suit :

```
elet : : (Eq a) => Tree a -> Trees a
elet tree trees = not (null (unbranch ((unit tree)<$>trees)))

unit : : Tree a -> Trees a
unit = ana un  where un x = [unode x]
```

Il est facile de définir une opération similaire sur les coalgèbres, c'est à dire une opération binaire $\langle * \rangle$ pour laquelle

$$ana\ (coalg_1\ \langle * \rangle\ coalg_2)\ =\ (ana\ coalg_1)\ \langle\$\rangle\ (ana\ coalg_2)$$

Cette opération généralise la construction qui produit l'automate fini dont le langage est l'intersection de deux langages réguliers à partir d'automates reconnaissant ces derniers :

```
infix 4 <*>

(<*>) : : (Eq a) => Coalg a b -> Coalg a c -> Coalg a (b,c)
(coalg1 <*> coalg2) (gen1,gen2) =
  [(a1,zip gens1 gens2) | (a1,gens1) <- coalg1 gen1,
                          (a2,gens2) <- coalg2 gen2,
                           a1 == a2 ]
```

4. Expansion d'une vue partielle

Afin de résoudre notre problème il nous reste juste à définir l'expansion d'une vue partielle comme une fonction qui retourne l'ensemble des arbres de dérivations ; cette fonction est donnée comme suit :

```
expansion : : (Eq b) => Gram a b -> (b->Bool) -> b -> [Tree b] -> [(Trees a,[Tree b])]
expansion  gram view axiom ders = parser axiom ders
where parser symb ders =
  if (view symb) then
    case ders of
      [] -> []
      (der :ders') -> if symb == (top der) then
                   [(Branch [(prod,list_trees) |
                            (prod,symbs) <- gram symb,
                             list_trees <- matches symbs (suc der)],
                    ders')]
                  else []
  else [(trees,ders2) |
        (ders1,ders2) <- split ders,
         trees <- [Branch [(prod,list_trees) |
                           (prod,symbs) <- gram symb,
                            list_trees <- matches symbs ders1]]]
  matches [] ders = if null ders then [[]] else []
  matches (symb :symbs) ders =
      [(trees : list_trees) | (trees,ders') <- parser symb ders,
                       list_trees <- matches symbs ders']
  split [] = [([],[])]
  split (x :xs) = [([],x :xs)]++[(x :xs1,xs2) | (xs1,xs2) <- split xs]
```

La fonction $expansion\ gram\ view\ ::\ b \rightarrow [Tree\ b] \rightarrow [(Trees\ a, [Tree\ b])]$ prend comme argument un symbole grammatical A puis une chaîne d'arbres de dérivation (ne contenant que des symboles visibles). Elle analyse cette chaîne d'entrée pour produire (de façon non-déterministe) un ensemble d'arbres de dérivation (représenté par un élément un élément de $Trees\ a$). Chacun de ces arbres de dérivation est issu du symbole grammatical A et sa projection selon la vue donnée ($view$) produit un préfixe de la chaîne de départ. On attache à chaque élément de $Trees\ a$ ainsi produit la liste d'arbres de dérivation résiduelle (c'est à dire n'ayant pas été consommée par le processus d'analyse). Un résultat de l'analyse est donc une paire de $(Trees\ a, [Tree\ b])$ c'est à dire constituée d'un ensemble d'arbres de dérivation et de la liste résiduelle correspondante. Le non déterminisme provient du fait qu'on produit une liste de tels résultats. Le type de cette fonction est ainsi semblable aux types des analyseurs syntaxiques de [6]. Cela aurait été intéressant, bien que sans incidence pratique particulière, de faire apparaitre cette fonction comme un anamorphisme ; c'est à dire de pouvoir associer une co-algèbre à chaque vue de sorte que l'expansion apparaisse comme l'anamorphisme associée à cette coalgèbre.

L'algorithme d'expansion peut être sensible à la façon dont est présentée la grammaire. Par exemple, et de façon classique, on a tout intérêt à factoriser les parties droites des productions d'un même symbole grammatical pour éviter la redondance de calculs. A l'inverse il ne sert ici à rien d'essayer d'éliminer les récursivités à gauche. En effet, par la nature même du problème, l'algorithme d'expansion n'a aucune raison de terminer car en général une vue partielle peut correspondre à un nombre infini de structures globales. Lorsque nous composons plusieurs telles fonctions par le combinateur $\langle\$\rangle$ le mécanisme

d'évaluation paresseuse les fait se comporter comme des co-routines de telle sorte que chacune d'entre elles va contraindre le comportement des autres en « élaguant » certaines branches dans leurs « trees » en cours de construction.

5. Conclusion

Nous avons apporté une solution au problème de la cohérence des vues dans les grammaires algébriques abstraites dans le cas où une vue est assimilée à un sous-ensemble de catégories syntaxiques. Cette solution repose sur le codage d'ensembles (infinis) d'arbres de dérivation par une structure de données paresseuse. Il s'agit donc d'un exemple d'utilisation de structures co-inductives et de co-algèbres ce qui est encore assez peu répandu en programmation surtout lorsque l'on songe que les notions duales de structures inductives et algèbres sont omniprésentes en programmation fonctionnelle. Or il se trouve que c'est dans la manipulation des structures co-inductives que le mécanisme d'évaluation paresseuse présente tout son intérêt et son originalité, c'est d'ailleurs pourquoi on utilise l'expression de structures de données paresseuses pour qualifier les structures de données co-inductives. Notre algorithme présente ainsi un exemple original d'utilisation du mécanisme d'évaluation paresseuse.

La partie centrale de notre algorithme, donnant l'expansion d'une vue partielle, est par ailleurs une utilisation d'un analyseur syntaxique fonctionnel [6]. Néanmoins un lexème est ici un arbre alors que tous les analyseurs fonctionnels que nous avons rencontrés dans la littérature ont pour lexèmes de simples symboles n'ayant aucune structure interne exploitée par l'algorithme. Nous pensons qu'il y a là matière à reflexion sur cette algorithmique des analyseurs fonctionnels.

Notre solution est particulièrement simple : son code en Haskell a été intégralement donné dans le corps de cet article. Il a été testé sur quelques exemples dont celui indiqué plus haut. La réponse a toujours été instantannée mais il vrai que nous ne l'avons pas utilisé avec des grammaires de tailles importantes. La preuve formelle de cet algorithme est la suite naturelle de ce travail.

Dans la poursuite de ce travail nous nous intéresserons également à l'ajout d'attributs et de dépendances fonctionnelles entre attributs, c'est à dire à l'extension de ce travail au cadre des grammaires attribuées. Il n'y a là *a priori* pas d'obstacles majeurs si on se repose en particulier sur la traduction des grammaires attribuées en programmes fonctionnels [9, 5, 1]. Cette extension peut être très intéressante d'un point de vue pratique car un attribut d'un symbole grammatical visible peut dépendre de la valeur d'attributs de symboles non visibles. Les attributs apparaissent ainsi comme des moyens de communication entre les différentes vues partielles. Les grammaires attribuées pourraient ainsi se présenter comme un formalisme de composition de composants réactifs (du fait de l'évaluation paresseuse).

A partir d'une grammaire abstraite il n'est pas trop difficile de concevoir un éditeur pour les structures correspondantes en utilisant la structure de zipper associée [8]. Les opérations de navigation s'implémentent aisémment avec la structure de zipper, on y adjoint des opérations de copier-coller et des fonctions d'édition associées à chaque production de la grammaire. Le quotient d'une grammaire par une vue cependant n'est plus une grammaire algébrique car on y trouve un nombre infini de productions. Nous avons un algorithme, non présenté ici, qui génère à partir d'une grammaire et d'une vue une représentation paresseuse de l'ensemble infini des productions dérivées. Dans l'avenir nous comptons exploiter cette structure afin de dériver un éditeur pour l'ensemble des vues partielles correspondantes.

6. Bibliographie

[1] Kevin S. Backhouse. A functional semantics of attribute grammars. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science, Springer-Verlag, 2002.

[2] C. Courbis, P. Degenne, A. Fau, D. Parigot. L'apport des technologies XML et objets pour un générateur d'environnements : SmartTools. *Revue Objets*, vol 9/3 :65-93, 2003

[3] Krzysztof Czarnecki, Ulrich W. Eisenecker. *Generative Programming : Methods, Tools, and Applications*. Addison Wesley, 2000.

[4] Sergey Dimitriev. Language Oriented Programming : The Next Paradigm. http ://www.onboard.jetbrains.com/articles/04/lop/index.html

[5] M. Fokkinga, J. Jeuring, L. Meertens, E. Meijer. A translation from Attribute Grammars to Catamorphisms. *The Squiggolist*, 2(1) :20-26, 1991.

[6] 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.

[7] Martin Fowler. Language Workbenches : The Killer-App for Domain Specific Languages. http ://www.martinfowler.com/articles/languageWorkbench.html

[8] G. Huet. The Zipper. *Journal of Functional Programming* 7(5), Sept 1997, pp. 549-554.

[9] T. Johnsson. Attribute grammars as functional programming paradigm. In G. Kahn, ed, *Proc. of 3rd Int. Conf. on Functional Programming and Computer Architecture*, FPCA'87, vol. 274 of Lecture Notes in Computer Science, 154-173, Springer-Verlag, 1987.

[10] Donald E. Knuth. Semantics of context free languages. *Mathematical System Theory*, 2(2) : 127-145, June 1968.

[11] J. Paakki. Attribute grammar paradigms–A high-level methodology in language implementation. *ACM Computing Surveys*, 27(2) : 196-255, June 1995.

[12] Charles Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In B. Randell (Ed.) The future of Software, Proceeding of the joint International Computer Limited. University of Newcastle seminar (Also : Technical Report MSR-TR-95-52, Microsoft Research, redmond), 1995.

[12] Eric Van Wyk, Oege de Moor, Ganesh Sittampalam, Ivan Sanabria Piretti, Kevin Backhouse, Paul Kwiatkowski. Intentional Programming : A Host of Language Features. Oxford University Computing Laboratory, PRG-RR-01-21, 2001.