UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
MARIE ANASTACIO

CONSTRUCTION DE CHAÎNES DE TRAITEMENT POUR L'ANALYSE DE TEXTE : UNE APPROCHE COMBINATOIRE TYPÉE

SEPTEMBRE 2015

Résumé

L'ingénierie de la langue se retrouve a l'intersection de plusieurs disciplines telles que la linguistique, la sémiologie, la philosophie, l'informatique, l'intelligence artificielle, etc. Malgré leurs succès, les technologies développées ont d'importantes limitations. Elles offrent un ensemble limité de fonctionnalités et sont souvent limitées en terme de communication avec d'autres logiciels. Elles rendent ainsi difficile la collaboration recherchée par les experts et leurs médiateurs. De plus, alors qu'elles sont rarement satisfaisantes comparées à l'ensemble des traitements possibles et utiles, intégrer une nouvelle fonctionnalité implique souvent de reconstruire l'application complète.
L'ensemble de ces problèmes entraine une redondance dans les sujets de recherche. En effet, les chercheurs développent des outils similaires puisque la réutilisation des logiciels existants est difficile. Les obstacles principaux sont les droits de propriété, qui ne permettent pas l'accès au code et ne rendent donc pas possible l'accès aux sources, et l'architecture monolithique de certaines applications.
Dans le présent projet, nous cherchons une modélisation formelle qui permet de supporter les besoins exprimés pas les ingénieurs de la langue. Notre défi est d'utiliser les systèmes applicatifs et la logique combinatoire pour manipuler des modules sous forme d'exécutables issus de différentes plates-formes et de langages distincts. Nous pourrons ainsi développer et expérimenter une multitude de chaînes de traitement et réutiliser des outils existants.

Remerciements

Je veux d'abord remercier tous ceux qui m'ont permis de m'intéresser au domaine de la recherche, en particulier mes professeurs de l'Université de Technologie de Belfort Montbéliard et Gillian Basso.

Je remercie également ceux qui ont transformé cet intérêt en un réel projet et qui m'ont offert l'opportunité de faire cette recherche, le professeur Ismail Biskri et l'Université du Québec à Trois-Rivières.

Finalement, je ne peux pas oublier les amis qui m'ont soutenue et parfois aidée en m'offrant un regard différent sur les problèmes que je rencontrais. Je pense ici à Emeric Garon, Jérémie Gobeil et Jérome De Wouters.

Enfin merci à ceux qui s'intéresseront et liront les articles qui ont été publié en parallèle de ce mémoire ou bien ce mémoire lui même. C'est l'idée que ce sur quoi je travaille puisse être utilisé par d'autres qui donne tout leur sens aux efforts que j'ai pu fournir.

Table des matières

Table des figures

Introduction

L'analyse des langues naturelles est un domaine qui se trouve au carrefour de disciplines diverses : la linguistique, la terminologie, la sémiologie, mais également des domaines tels que la logique, l'informatique, l'intelligence artificielle, etc. Ceux-ci vont eux même en chercher d'autres, tels que la psychologie ou la philosophie. Les chercheurs impliqués sont aussi nombreux que les expertises demandées, et les outils développés afin d'appliquer les techniques imaginées par les chercheurs se multiplient. Chacun offre des fonctionnalités différentes, mais n'est que rarement satisfaisant. En effet, ils ne sont pas complets quant aux fonctionnalités implémentées. Et alors que celles-ci se multiplient, leur ajout dans un programme peut demander une refonte complète de l'architecture du logiciel. Vouloir déléguer une partie des traitements à une autre application se heurte aux limites de la communication entre deux programmes, souvent développés par des équipes différentes. Ainsi la collaboration est difficile et les mêmes travaux sont répétés plusieurs fois, entrainant une redondance des sujets de recherche, et donc une perte de temps et d'argent. À ces problèmes s'ajoutent deux autres obstacles. D'une part, la propriété intellectuelle empêche l'accès au code et limite donc la réutilisabilité des outils développés, d'autre part les différentes applications développées dans des paradigmes, des langages, et des environnements différents sont souvent très limitées en terme de compatibilité. Dans ce contexte, chaque modification effectuée au traitement appliqué doit passer par une nouvelle phase de développement par des informaticiens avant d'obtenir de nouveaux résultats. Ceux-là mêmes qui amèneront peut-être à modifier à nouveau les

traitements appliqués. Le processus d'essai et d'erreur, qui est une base du travail de recherche, est considérablement ralenti par ces allées et venues.

La présente recherche a pour but de proposer des pistes de solution à ces problèmes. C'est-à-dire de permettre aux différents logiciels de travailler ensemble, et d'éviter le retour en développement de ces outils, en proposant aux utilisateurs d'organiser eux même les fonctionnalités qui leur sont proposées. Tout d'abord, puisqu'il n'est pas possible, ni même souhaitable, de forcer les institutions à diffuser librement le code source de leur travail, une solution serait de faciliter la communication entre les différentes fonctionnalités. Aussi, l'utilisateur doit être au centre de notre réflexion et pouvoir réorganiser ses outils. Pour ce second point, les chaînes de traitements sont actuellement utilisées par des plateformes, D2K/T2K [10] ou Knime [21] par exemple, qui offrent la flexibilité recherchée. Le défi principal reste alors de lier les éléments de ces chaînes de traitement et de leur donner un cadre dans lequel travailler ensemble. Un travail précédent effectué pas Marc-André Rochette [18] a montré que les liens entre les opérations d'une chaîne de traitement pouvaient être exprimés grâce à la logique combinatoire. Celle-ci permet de représenter l'ordre d'exécution des modules de la chaîne. Pour ajouter la gestion des types de données, le présent travail utilisera les systèmes applicatifs. Ceci sans revenir sur le travail précédent, mais en les ajoutant au concept déjà développé. Les systèmes applicatifs et la logique combinatoire sont déjà largement utilisés en traitement des langues naturelles. Le modèle choisi est basé sur celui présenté en 1997 par Ismail Biskri et Jean-Pierre Desclés [5].

Le premier chapitre présentera le paysage technologique actuel ainsi que les logiciels existants qui tentent de répondre au problème soulevé. Il sera suivi d'une description des outils utilisés dans la formalisation ici proposée. C'est-à-dire, comme exprimé précédemment, les systèmes applicatifs et la logique combinatoire. Le troisième chapitre expliquera la méthodologie, ainsi que les règles et algorithmes employés.

Pour finir, les règles et algorithmes ont été implémentés et les résultats de tests seront présentés. Les résultats permettront de voir si les techniques choisies sont intéressantes.

Chapitre 1

État de l'art

L'objectif de ce chapitre est de décrire la situation actuelle dans ses grandes lignes, et de placer le présent travail dans ce contexte. Les concepts de paradigme de programmation, langage de programmation et chaîne de traitement seront présentés sommairement. Les deux premiers, car ils sont les piliers de la programmation actuelle, et le dernier, car il est l'outil choisi afin de permettre à des utilisateurs non informaticiens d'organiser les opérations qu'ils veulent faire effectuer par la machine.

1.1 Paradigmes et langages de programmation

La programmation permet de résoudre des problèmes. Le programmeur commence par formaliser ce qu'il doit résoudre, puis donne à la machine une série d'instructions qui permettront d'atteindre un résultat, une solution à ce problème. Pour ce faire, il utilise les langages de programmation, qui servent d'interface entre le langage humain et le langage machine. Aussi, puisqu'il y a plusieurs manières de conceptualiser cette communication homme-machine, chaque langage peut être catégorisé dans un

ou plusieurs paradigmes.

1.1.1 Paradigmes

En 1962, Kuhn définit ce qu'est un paradigme [2].

Définition 1 (Paradigme). *Réalisations scientifiques universellement reconnues qui, pour un temps, fournissent un modèle de problèmes et solutions à une communauté de praticiens.*
(traduction libre)

Ces paradigmes peuvent être définis en effectuant une description de leurs caractéristiques, ou en prenant une liste d'exemples qui, tous ensemble, représentent ces caractéristiques. Puisque la programmation s'applique à la résolution de problèmes, la définition précédente peut être affinée. En particulier, D. Appleby cite Wegner qui, en 1988, fait référence aux paradigmes comme à des « schémas de pensée pour la résolution de problèmes » .
Les paradigmes de programmation sont nombreux et leur énumération exhaustive ne serait pas utile au propos. Ils se divisent en deux grandes familles : les paradigmes impératifs et les paradigmes déclaratifs.

Présentation des deux types de paradigmes Les paradigmes impératifs regroupent un grand nombre de paradigmes qui sont parmi les plus utilisés actuellement. Lors de la programmation impérative, le problème original est découpé en une séquence d'instructions que la machine devra effectuer afin d'obtenir un résultat. La programmation impérative pousse à se poser la question « comment arriver au résultat ? » puisque chaque opération doit être décrite par l'utilisateur.
Programmer selon ce paradigme correspond à indiquer à la machine chaque modification qu'elle devra faire à sa mémoire. Celle-ci effectue ainsi une série de calculs pour

résoudre le problème.

Les paradigmes déclaratifs sont moins répandus, bien que le paradigme fonctionnel soit très utilisé. Plutôt que sur le moyen d'obtenir un résultat, ils se concentrent sur les éléments qui permettent d'y arriver. Ces paradigmes se basent sur des théories mathématiques telles que la logique, ou les fonctions. Ils représentent ces éléments et leurs interactions, mais décrivent plus les éléments qui permettent de résoudre le problème que les calculs à faire pour y parvenir.
Par définition, ils sont donc plus éloignés de la machine et n'utilisent pas de variables.

1.1.2 Langages

Définition 2 (Langage (informatique)). *Ensemble organisé de symboles, de mots-clés, de caractères et de règles (instructions et syntaxe) utilisé pour adresser des commandes à l'ordinateur et assurer la communication avec la machine.*
[15]

Puisque le langage humain n'est pas compris par l'ordinateur, et que le langage machine est peu compréhensible aux humains, les langages de programmation sont les liens de traduction entre l'homme et la machine. Et tout comme il existe plusieurs manières de conceptualiser cette communication (les paradigmes), il existe plusieurs langages, chacun rattaché à un ou plusieurs paradigmes de programmation.
Le langage de programmation détermine la syntaxe utilisée pour indiquer les opérations que l'ordinateur devra effectuer.

1.2 Interopérabilité et compatibilité

Plusieurs paradigmes, plusieurs langages, la multiplicité des moyens impose de faire des choix, et travailler avec plusieurs langages ou paradigmes n'est pas simple. Chacun offre des avantages et des inconvénients. Programmer demande au préalable d'évaluer quel est le langage qui répond le mieux aux besoins exprimés. Certains demandent en plus d'allier plusieurs environnements physiques, plusieurs types de médias, plusieurs systèmes d'exploitation. Souvent, plutôt que de recoder totalement des fonctionnalités existantes, des composants sont repris dans d'autres programmes, ou bien des librairies sont utilisées. Ceux-ci existent déjà et permettent de gagner du temps, mais sont également codés dans un langage précis. Ainsi apparait le besoin de faire travailler des langages les uns avec les autres.
Les différents langages travaillent plus ou moins bien ensemble de manière native. Certains sont vraiment difficiles à allier, alors que d'autres sont pensés pour être utilisés de concert.

Définition 3 (Compatibilité). *Capacité d'un ordinateur et du matériel associé à communiquer entre eux et à échanger des données, ou de deux ordinateurs à exécuter les mêmes programmes sans que les résultats attendus soient compromis ou altérés.*
[15]

Deux programmes compatibles peuvent communiquer ensemble. Dans le meilleur des cas, ils peuvent travailler ensemble et échanger des informations en temps réel, mais la compatibilité couvre également des programmes qui peuvent écrire et lire un même format de fichier. Lorsque deux langages, ou deux programmes, doivent agir ensemble, la compatibilité est le minimum requis. La majorité des langages essaient donc d'offrir une compatibilité minimale avec les autres, mais il est fréquent qu'elle ne soit pas suffisante.

Définition 4 (Interopérabilité). *Capacité que possèdent des systèmes informatiques*

hétérogènes à fonctionner conjointement, grâce à l'utilisation de langages et de protocoles communs, et à donner accès à leurs ressources de façon réciproque.
[15]

Deux systèmes interopérables peuvent travailler ensemble de manière transparente. Ils communiquent directement, et travaillent ensemble. Cela repose souvent sur une accessibilité et une visibilité importante de l'un sur l'autre. Ils sont ainsi faits pour travailler ensemble. Certains programmes peuvent, par exemple, ouvrir et modifier en même temps un même fichier. Chacun voit alors en temps réel les modifications effectuées par l'autre.

1.3 Chaînes de traitement

Plusieurs domaines utilisent le concept de « Chaîne de traitement ». Le principe général est assez simple : les traitements qui vont être effectués sont représentés par une série d'opérations qui se suivent les unes les autres.

1.3.1 Définition

Définition 5 (Chaîne de traitement). *Ensemble de travaux « enchaînés » exécutés à la suite les uns des autres et considérés comme une partie d'une application.*
[15]

Les chaînes de traitement sont utilisées en informatique, mais également dans d'autres domaines. Elles ont l'avantage de ne pas demander de connaissances spécifiques

pour être utilisées et comprises. De ce fait, plusieurs programmes permettent leur utilisation.

1.3.2 Plateformes existantes

Plusieurs logiciels existants utilisent les chaînes de traitement comme moyen de permettre aux utilisateurs de programmer sans avoir besoin de code.

— Knime [14, 21]

Celui-ci permet la création d'un ou plusieurs workflows qui permettent d'enchaîner des actions ou calculs dans un ordre donné. Ce logiciel est utilisé dans divers domaines. Les nœuds de la chaîne de traitement sont codés en Java.

— RapidMiner [17]

Logiciel de data-mining bien intégré sur diverses plateformes. L'utilisateur crée des chaînes de traitement à partir de modules qui sont codés en java.

— Gate [11, 7]

Il permet de faire du traitement linguistique en intégrant un langage de script. Il ne fonctionne pas à base de modules et demande donc à l'utilisateur d'avoir quelques connaissances en programmation pour pouvoir être utilisé.

— T2K-D2K [9, 10]

Proche de Knime dans son utilisation et son fonctionnement, les modules sont également codés en Java.

— Orange [16]

Il s'agit d'un logiciel d'analyse et de visualisation de données. Des modules peuvent être programmés en Python et reliés les uns aux autres pour choisir quels traitements seront appliqués aux données et quel type de visualisation est désirée.

— UIMA [1]

Contrairement aux précédents, il ne s'agit pas d'un logiciel mais d'un framework. C'est-à-dire que pour l'utiliser il faut coder. Les fonctionnalités sont

offertes sans interface utilisateur.

Tous ces logiciels, bien qu'ils permettent de créer des chaînes de traitement, se cantonnent à un seul langage de programmation. De plus, même lorsque deux plateformes utilisent le même langage de programmation, un module codé pour l'un ne sera pas utilisé par l'autre, car tous ont des spécifications qui leur sont propres. Donc même s'il permet à un utilisateur n'ayant pas de grandes connaissances en programmation de construire des chaînes de traitement, le problème de la compatibilité entre les différents langages reste entier.

Cependant, leur mode de fonctionnement appuie le fait que les chaînes de traitement sont un outil connu et utilisé qui permet de rendre notre outil accessible au plus grand nombre.

Chapitre 2

Outils

Ce chapitre présente les théories et objets mathématiques qui sont utilisés pour essayer de répondre à la problématique. Les systèmes applicatifs typés et la logique combinatoire sont utilisés en analyse des langues naturelles [20]. La logique combinatoire a d'ailleurs été utilisée pour construire une langue, le Lojban [6], qui tente de permettre une communication universelle entre les hommes, mais également avec les machines. Ainsi que pour construire des langages de programmation fonctionnels, tels que Haskell [13].

2.1 Systèmes applicatifs typés

Les systèmes applicatifs typés ressemblent aux fonctions usuelles mathématiques. Ils représentent des fonctions ayant une ou plusieurs entrées et une sortie unique, chaque entrée et chaque sortie étant typées.

Soit f une fonction dont l'entrée est de type Int et la sortie de type $String$. On

notera $f : FIntString$. Soit un entier n. L'application de la fonction f à cet entier peut s'écrire de plusieurs manières. La notation qui sera utilisée est la notation préfixée : $f\ n$. Soit la fonction $g : FIntFIntString$. Elle s'applique à deux entiers et renvoie une chaîne de caractères. L'application de g à n et à un second entier m sera notée $f\ n\ m$. Plutôt qu'une fonction prenant deux entrées, la fonction g peut être également interprétée comme une fonction qui prend un entier en entrée et retourne une autre fonction de type $FIntString$.

Curryfication La curryfication d'une fonction est l'opération consistant à en créer une nouvelle à partir d'une existante qui a plusieurs entrées.

Définition 6 (Currification). *Soient t_1, t_2 et t_3 des types de données. Soit f : $Ft_1Ft_2t_3$. Soient x et y deux variables de types respectifs t_1 et t_2. Les fonctions curryfiées à partir de f avec ces variables sont :*

- *f_y telle que $f_yx = fxy$*
- *f_x telle que $f_xy = fxy$*

Soient n et m deux valeurs de types respectifs t_1 et t_2. Les fonctions curryfiées à partir de f avec ces valeurs sont :

- *f_n telle que $f_nx = fxn$*
- *f_m telle que $f_my = fmy$*

Exemple : *$fxy = x + y$ avec x et y des entiers.*

$f_yx = x + y$ et $f_xy = x + y$

On peut également fixer une des deux valeurs de l'addition : $f_5x = x + 5$

La curryfication permet de créer une fonction à partir d'une fonction et de variables. Ce principe peut également être utilisé avec les combinateurs pour créer un combinateur à partir d'un existant.

2.2 Logique combinatoire

En mathématiques, une fonction est associée à des types. On la définit comme une application ayant un ensemble de départ et un ensemble d'arrivée. Le λ-calcul met de côté la notion de type et définit des fonctions qui n'ont pas besoin d'être décrites par le type d'entrée et de sortie. Le paradigme fonctionnel, qui est de type déclaratif, est basé sur ce type de calcul. Cependant, elle reste collée au besoin d'avoir des variables. Schönfinkel introduit la logique combinatoire en 1924 [19], suivi par Curry et Feys en 1958 [8]. Celle-ci s'abstrait de ce besoin, elle fonctionne sans variables.

2.2.1 Principes fondamentaux de la logique combinatoire

Les combinateurs sont des opérateurs qui agissent sur les éléments qui les suivent. Ces éléments peuvent être des combinateurs, des arguments ou des fonctions. Pour bien comprendre les lois qui les régissent, le plus simple est de commencer par le lambda calcul.

2.2.2 Du lambda calcul à la logique combinatoire

En 1941 apparaît le λ-calcul, qui permet de s'abstraire du type des variables. Il est utilisé dans le cadre de la programmation afin de représenter des fonctions pour lesquelles les entrées peuvent avoir plusieurs types. Pour ce faire, le λ-calcul met de côté le principe de type et travaille sur des λ-expressions. Par exemple, l'opération $x - y$ peut être représentée par les λ-expressions $\lambda x.x - y$, qui a x associe $x - y$, et $\lambda y.x - y$, qui a y associe $x - y$. Elle peut également être représentée par $\lambda xy.x - y$.

Pour atteindre un niveau d'abstraction supérieur, Schönfinkel introduit les com-

binateurs [19]. Alors que les λ-expressions se définissent selon une variable, les combinateurs n'en ont pas besoin. Schönfinkel introduit divers combinateurs, ceux qui sont à la base de tous les autres sont les combinateurs I (identité),K (constant) et S(fusion). En effet, il montre que tous les combinateurs peuvent s'exprimer comme des combinaisons de ces opérateurs de base.

2.2.3 Propriétés importantes

Hindley et al. [12] présentent une introduction aux propriétés de la logique combinatoire sur laquelle se base le présent chapitre.

La réduction d'une expression peut se définir à partir des axiomes de réduction des trois opérateurs de base I, K et S. Pour x, y et z des opérandes,

(I) $Ix \equiv x$

(S) $Sxyz \equiv xz\,(yz)$

(K) $Kxy \equiv x$

(ρ) $x \equiv x$

Aussi, pour X, Y et Z des expressions combinatoires,

(a) $(X \equiv X') \Rightarrow (ZX \equiv ZX')$

(b) $(X \equiv X') \Rightarrow (XZ \equiv X'Z)$

(c) $(X \equiv Y$ et $Y \equiv Z) \Rightarrow (X \equiv Z)$

Soient, par exemple, w, x, y et z des fonctions et opérandes. Soit l'expression $KSwxyz$. Pour la réduire, le combinateur K sera appliqué en premier, suivi du combinateur S.

$$KSwxyz \equiv Sxyz$$

$$Sxyz \equiv xz(yz)$$

Théorème de Church-Rosser *Soit X, Y et W des expressions combinatoires telles que :*

- $X \equiv Y$
- $X \equiv W$

Alors il existe une expression combinatoire W telle que :

- $Y \equiv Z$
- $W \equiv Z$

Ce théorème signifie que la forme réduite d'une expression combinatoire est unique. On appelle cette forme la forme normale.

Puissance et distance d'un combinateur

La puissance d'un combinateur La puissance d'un combinateur est notée en exposant. Elle permet d'augmenter le nombre d'opérandes concernés par l'action du combinateur. Elle est définie en utilisant le combinateur B. B a pour règle de réduction : $Bxyz \equiv x\,(yz)$. Il sera détaillé plus dans la section suivante.

Définition 7 (Puissance). *Soit X un combinateur. Le combinateur X de puissance n sera noté X^n tel que*

- $X^1 \equiv X$
- $X^n \equiv BXX^{(n-1)}$

Exemple : $K^2xyz \equiv BKKxyz \equiv K(Kx)yz \equiv (Kx)z \equiv Kxz \equiv x$

La distance d'un combinateur La distance d'un combinateur est notée en indice. Elle permet de décaler l'application de son action vers des éléments plus lointains.

Définition 8 (Distance). *Soit X un combinateur. Le combinateur X de distance n sera noté X_n tel que*

- $X_0 \equiv X$

— $X_n \equiv B^n X$

Exemple : $S_1 wxyz \equiv BSwxyz \equiv S(wx)yz \equiv wxz(yz)$

2.2.4 Autres combinateurs

Si toutes les opérations peuvent être définies avec les combinateurs K, S et I, n'utiliser qu'eux trois rendrait les expressions peu lisibles. D'autres combinateurs ont donc été introduits.

Le combinateur B de composition

Soient x, y et z des expressions combinatoires quelconques. Le combinateur B est défini par la réduction : $Bxyz \equiv x\,(yz)$
Il est appelé combinateur de composition. En terme fonctionnel, il correspond à la composition de deux fonctions. Par exemple, si f et g sont des opérateurs et x un opérande, Bfg est un opérateur complexe qui s'applique à x et équivaut à appliquer g à x puis à appliquer f au résultat. En terme fonctionnel : $f(g(x))$.

Comme dit précédemment, le combinateur B peut s'exprimer en tant que combinaison de S et K : $S\,(KS)\,K \equiv B$

$$\begin{aligned} S\,(KS)\,Kxyz &\equiv (KS)\,x\,(Kx)\,yz \\ &\equiv KSx\,(Kx)\,yz \\ &\equiv S\,(Kx)\,yz \\ &\equiv (Kx)\,z\,(yz) \\ &\equiv Kxz\,(yz) \\ &\equiv x\,(yz) \end{aligned}$$

$$S\,(KS)\,Kxyz \equiv Bxyz$$

$$S\,(KS)\,K \equiv B$$

Le combinateur C de permutation

Soient x, y et z trois expressions combinatoires quelconques. Le combinateur C est défini par la réduction : $Cxyz \equiv xzy$
Il est appelé combinateur de permutation, car il permet de permuter deux éléments. Par exemple, si x est un opérateur à deux opérandes, Cx correspond à appliquer l'opération x en échangeant ses deux entrées.

Le combinateur C peut s'exprimer en tant que combinaison de S et K : $S\,(BBS)\,(KK) \equiv C$

$$\begin{aligned} S\,(BBS)\,(KK)\,xyz &\equiv (BBS)\,x\,((KK)\,x)\,yz \\ &\equiv B\,(Sx)\,(KKx)\,yz \\ &\equiv (Sx)\,(KKxy)\,z \\ &\equiv xz\,(KKxyz) \\ &\equiv xz\,(Kyz) \\ &\equiv xzy \end{aligned}$$

$$S\,(BBS)\,(KK)\,xyz \equiv Cxyz$$

$$S\,(BBS)\,(KK) \equiv C$$

Le combinateur W de duplication

Soient x et y deux expressions combinatoires quelconques. Le combinateur W est défini pas la réduction : $Wxy \equiv xyy$
Il est appelé combinateur de duplication, car il permet de créer un double d'un élément. Par exemple, si f est un opérateur à deux entrées, Wf est un opérateur à une seule entrée qui correspond à f auquel on passe deux fois la même entrée.

Le combinateur W peut s'exprimer en tant que combinaison de S et K : $SS(KI) \equiv W$

$$\begin{aligned} SS(KI)\,xy &\equiv Sx(KIx)\,y \\ &\equiv xy(KIxy) \\ &\equiv xy(Iy) \\ &\equiv xyy \end{aligned}$$

$$SS(KI)\,xy \equiv Wxy$$
$$SS(KI) \equiv W$$

Le combinateur Φ de coordination

Soient w, x, y et z des expressions combinatoires quelconques. Le combinateur Φ est défini pas la réduction : $\Phi wxyz \equiv w(xz)(yz)$
Il permettra de distribuer un opérande entre deux opérateurs.

Le combinateur Φ peut s'exprimer en tant que combinaison de S et K, mais pour

simplifier son expression, le combinateur B sera également utilisé : $B\,(BS)\,B \equiv \Phi$

$$\begin{aligned} B\,(BS)\,Bwxyz &\equiv (BS)\,(Bw)\,xyz \\ &\equiv S\,(Bwx)\,yz \\ &\equiv (Bwx)\,z\,(yz) \\ &\equiv w\,(xz)\,(yz) \end{aligned}$$

$$B\,(BS)\,Bwxyz \equiv \Phi wxyz$$

$$B\,(BS)\,B \equiv \Phi$$

Le combinateur Ψ de distribution

Soient w, x, y et z des expressions combinatoires quelconques. Le combinateur Psi est défini pas la réduction : $\Psi wxyz \equiv w(xy)(xz)$
Son action est proche de celle de Φ mais au lieu de distribuer les opérandes, il distribue les opérateurs.

Le combinateur Psi peut s'exprimer en tant que combinaison de S et K, mais les combinateurs B et Φ seront utilisés pour simplifier cette expression : $\Phi\,(\Phi\,(\Phi B))\,B\,(KK) \equiv \Psi$

$$\begin{aligned} \Phi\,(\Phi\,(\Phi B))\,B\,(KK)\,wxyz &\equiv \Phi\,(\Phi B)\,(Bw)\,(KKw)\,xyz \\ &\equiv \Phi B\,(Bwx)\,(KKwx)\,yz \\ &\equiv B\,(Bwxy)\,(KKwxy)\,z \\ &\equiv Bwxy\,(KKwxyz) \\ &\equiv w(xy)\,(Kxyz) \\ &\equiv w(xy)(xz) \end{aligned}$$

$$\Phi\left(\Phi\left(\Phi B\right)\right)B\left(KK\right)wxyz \equiv \Psi wxyz$$

$$\Phi\left(\Phi\left(\Phi B\right)\right)B\left(KK\right) \equiv \Psi$$

2.2.5 Association des systèmes applicatifs et de la logique combinatoire

Les systèmes applicatifs permettent d'exprimer des fonctions et leur enchaînement. Ils permettent de décrire l'application de plusieurs fonctions à la suite selon une chaîne de traitement. Par exemple, soient f une fonction qui s'applique sur deux entrées et g une fonction qui s'applique sur une seule entrée. Soient x, y des valeurs. Si ces fonctions s'enchaînent de telle sorte que la valeur renvoyée par g appliqué à x est utilisée comme première entrée par f, elle s'exprimera ainsi : $f(gx)y$. Ce type d'expression, bien que lisible par l'homme, a pour désavantage de mélanger les noms des fonctions et leurs entrées.

Les combinateurs permettent d'exprimer autrement cet enchaînement de fonctions. Cette expression peut aussi s'écrire : $Bfgxy$. Cette nouvelle expression est séparable en trois parties distinctes : les combinateurs, B, les fonctions, fg, et les valeurs d'entrées, xy. Les deux premières parties représentent la chaîne de traitement et l'ordre d'application de ses fonctions, alors que la dernière donne l'ensemble des entrées attendues pour cette chaîne.

Ainsi associées, ces deux théories permettent de développer le modèle qui sera présenté dans le chapitre suivant.

Chapitre 3

Méthodologie

Maintenant que les théories utilisées ont été présentées, ce chapitre décrira la méthodologie utilisée afin de représenter les liens entre les modules et leur ordre d'exécution. Il commencera par un rappel des objectifs et contraintes de ce travail, qui sera suivi des règles qui ont été mises en place pour atteindre ces objectifs.

3.1 Objectifs et contraintes

Comme évoqué précédemment, notre objectif principal est de permettre à des programmes, ou scripts, de travailler de concert sans avoir besoin de leur imposer des contraintes, sans avoir accès à leur code, et sans que l'utilisateur n'ait besoin de connaissances en programmation. Les programmes doivent donc communiquer en toute liberté quant à leur paradigme, leur langage, leur fonctionnement, et sans avoir à dévoiler ces informations.

Le seul élément qu'ils doivent avoir en commun est de prendre une ou plusieurs entrées et d'avoir une sortie. Nous les appellerons des modules. Ici interviennent les fonctions

applicatives. En effet, chaque module peut être représenté par une fonction applicative dont les entrées et sortie sont celles du module. Mais l'exécution de chaque module n'est pas totalement indépendante des autres, et en particulier des modules dont l'exécution détermine ses entrées. Les combinateurs gardent une trace des opérations et de leur ordre, ce qui permet de reconstruire une chaîne de traitement à partir du résultat de son analyse.

Exemple Un module prenant en entrée un texte et retournant une métrique le concernant sera représenté par une fonction d'un texte vers un réel. Notons M_1 ce module, Txt le type texte et R le type réel.
Le module pourra être noté $M_1 : F_{TxtR}$ et schématisé à la figure 3.1.

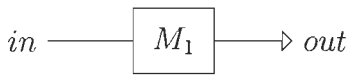


FIGURE 3.1 – Représentation d'un module

L'utilisation des fonctions applicatives typées permet ici de représenter toutes les informations utiles du module considéré.

Soit un second module M_2, qui prend en entrée deux métriques d'un texte et renvoie une phrase le représentant. Ce module peut s'écrire $M_2 : F_R F_{RTxt}$ et est représenté par la figure 3.2

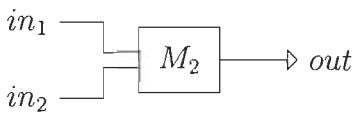


FIGURE 3.2 – Représentation d'un module à deux entrées

Considérons à présent que la valeur qui sera prise par la première entrée du module M_2 est celle qui est renvoyée par le module M_1. Cet enchaînement est représenté par la figure 3.3.

Le schéma de la figure 3.3 peut être exprimé par l'expression $out \equiv M_2(M_1 in_1) in_2$.

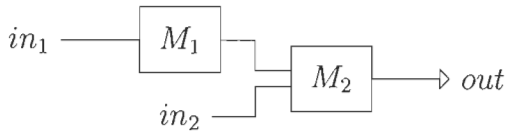


FIGURE 3.3 – Représentation d'une chaîne de traitement

Des combinateurs peuvent être introduits dans cette expression comme suit :

$$\begin{aligned} out &\equiv M_2(M_1 in_1) in_2 \\ &\equiv BM_2M_1 in_1 in_2 \end{aligned}$$

Aussi, si on pose $M \equiv BM_2M_1$, on obtient $out \equiv Min_1in_2$. Avec M un module à deux entrées. Notre chaîne a donc été réduite à un seul module.

Les combinateurs permettent d'exprimer simplement le lien entre les modules et gardent leur ordre d'exécution.

Traiter les chaînes à la main est aisé lorsqu'elles contiennent deux ou trois modules, mais de plus en plus long lorsque les chaînes se complexifient. La première chose à faire est donc de trouver un moyen d'exprimer les situations possibles par des règles. Une fois ces règles exprimées, l'enjeu suivant sera de choisir leur ordre d'application et de déterminer l'ensemble des cas qu'elles permettent de couvrir.

3.2 Définitions de règles formelles

Les règles sont basées sur des situations simples, mais permettent lorsqu'on les combine de couvrir un grand nombre de configurations.

3.2.1 Règle applicative

La première règle à définir est la règle applicative. Elle permet de représenter l'application d'un module à son entrée pour en calculer le résultat.

Soient x et y deux types de données différents ou non.
Soient X une variable de type x et M_1 un module exprimant une fonction de x vers y.

Règle 1. $$\frac{[X : x] \quad + \quad [M_1 : Fxy]}{[Y : y]}$$

3.2.2 Règle de composition

La règle de composition permet de réduire deux modules qui s'enchaînent en un seul module.

Soient x, y et z trois types de données différents ou non. Soient M_1 un module exprimant une fonction de x vers y et M_2 un module exprimant une fonction de y vers z. Si la valeur de sortie de M_1 est celle d'entrée de M_2. La sortie *out* de cette chaîne de traitement se calculera grâce à l'expression :

$$\begin{aligned} out &\equiv M_2(M_1 in) \\ &\equiv BM_2M_1 in \end{aligned}$$

La situation peut être représentée par le schéma 3.4.

Ce qui permet de formuler la règle de composition.

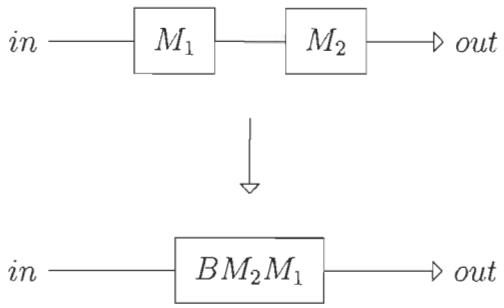


FIGURE 3.4 – Application de la règle de composition

$$\frac{[M_1 : Fxy] \quad + \quad [M_2 : Fyz]}{[(BM_2M_1) : Fxz]}\ \text{B}$$

Si la sortie de M_1 est utilisée par un autre module, celui-ci ne disparaîtra pas de la chaîne. En effet, si un module $M_3 : Fyw$ avait pour entrée la sortie de M_1, l'application de la règle de composition ne doit pas lui ôter son entrée. Cette situation est illustrée par le schéma 3.5.

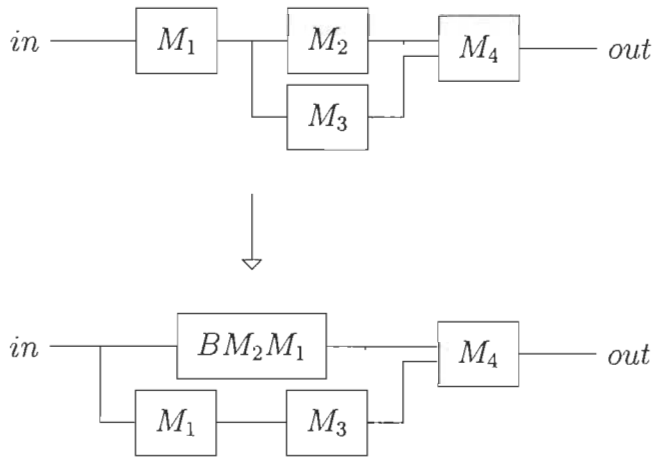


FIGURE 3.5 – Application de la règle de composition - cas particulier

Puisque les modules n'ont pas toujours une seule entrée, cette règle doit être généralisée à des modules ayant plusieurs entrées.

Soit M_1' un module ayant n entrées et une sortie. Chaque entrée in_i est de type x_i, différent ou non des autres entrées. La sortie est de type y. Sa sortie est l'entrée du module M_2. De la même manière que pour le précédent, ce problème peut s'exprimer

en terme de combinateurs.

$$\begin{aligned} out &\equiv M_2(M_1' in_1 \dots in_n) \\ &\equiv B^n M_2 M_1' in_1 \dots in_n \end{aligned}$$

Soient x_k les types respectifs des in_k pour $k \in [|1, n|]$.
La règle de composition peut être reformulée pour un module à plusieurs entrées.

$$\frac{[M_1 : Fx_1 \dots Fx_n y] \quad + \quad [M_2 : Fyz]}{[(B^n M_2 M_1) : Fx_1 \dots Fx_n z]}\ \text{B}$$

Si M_2 possède plusieurs entrées, la règle précédente reste inchangée. En effet, soit M_2' un module ayant k entrées et une sortie. Chaque entrée in_j' est de type y_j, différent ou non des autres entrées, et $y_1 = y$.

$$\begin{aligned} out &\equiv M_2(M_1' in_1 \dots in_n) in_2' \dots in_k' \\ &\equiv B^n M_2 M_1' in_1 \dots in_n in_2' \dots in_k' \end{aligned}$$

Ainsi, la règle générale peut-être exprimée :

Règle 2. $$\frac{[M_1 : Fx_1 \dots Fx_n y_1] \quad + \quad [M_2 : Fy_1 \dots F_m z]}{[(B^n M_2 M_1) : Fx_1 \dots Fx_n Fy_2 \dots Fy_m z]}\ B$$

3.2.3 Règle de composition distributive

La règle de composition distributive permet de fusionner un module qui précède la deuxième entrée d'un autre module. Soit $M1$ un module dont l'entrée est de type x et la sorti de type y, et $M2$ un module ayant deux entrées de types z et y et une sortie de type t. es modules sont agencés selon le schéma de la figure 3.6 et représentés par

l'expression combinatoire :

$$\begin{aligned} out &\equiv M_2 in(M_1 in) \\ &\equiv SM_2M_1 in \end{aligned}$$

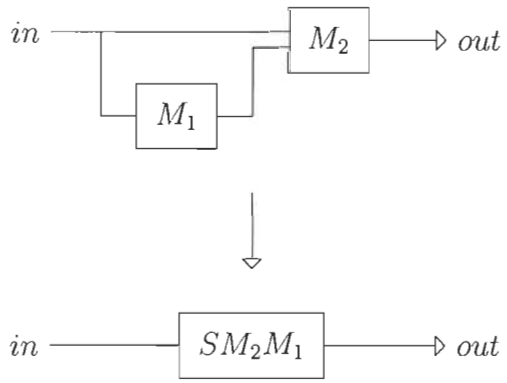


FIGURE 3.6 – Application de la règle de composition distributive

Ce qui permet de déduire la règle de composition distributive :

Règle 3. $$\frac{[M1 : Fxy] \quad + \quad [M2 : FxFyz]}{[(SM2M1) : Fxz]}\ S$$

Pour cette règle, il faut prendre en compte le même cas particulier que celui de la règle de composition exprimée en page 27.

3.2.4 Règle de duplication

Lorsque les entrées d'un module sont similaires, les fusionner semble une bonne pratique afin de diminuer la quantité d'information requise. Soit M un module qui prend deux entrées de type x et rend une valeur de type y. La fusion des deux entrées identique est faite par le combinateur W.

$$\begin{aligned} out &\equiv Minin \\ &\equiv WMin \end{aligned}$$

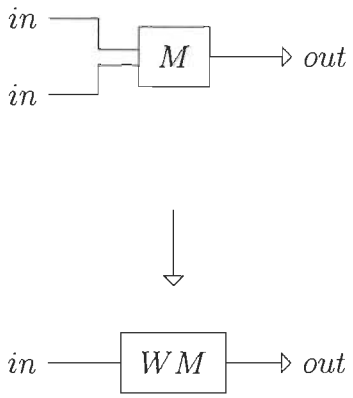


FIGURE 3.7 – Application de la règle de duplication

Ce qui amène la règle de duplication pour deux entrées, représentée par le schéma de la figure 3.7.

$$\frac{[\mathrm{M} : \mathrm{FxFxy}]}{[(\mathrm{WM} : \mathrm{Fxy}]}\ \mathrm{w}$$

Cependant, le nombre d'entrées identiques peut être plus grand que deux. Dans ce cas, la règle précédente devra être généralisée à un nombre quelconque d'entrées identiques. Soit M' un module acceptant n entrées identiques notées in de type x.

$$\begin{aligned} out &\equiv M'in...in \\ &\equiv W^n M'in \end{aligned}$$

Ce qui se traduit par la règle suivante.

$$\frac{[\mathrm{M} : (\mathrm{Fx...Fx})^{nfois}\mathrm{y}]}{[(\mathrm{W}^n\mathrm{M} : \mathrm{Fxy}]}\ \mathrm{w}$$

De la même manière que pour les autres, cette règle peut être appliquée même si le module possède plus d'une entrée. L'important est que les n premières soient identiques. Soit M un module ayant m entrées. Chaque entrée in_i est de type x_i, différent ou non des autres entrées, et $x_1 = x_2 = \cdots = x_n$.

Règle 4. $\dfrac{[M : (Fx...Fx)^{n\,fois}\, Fx_{n+1} \ldots Fx_m y]}{[(W^n M : FxFx_{n+1} \ldots Fx_m y]}\ W$

3.2.5 Règle de permutation

Les règles précédentes s'appliquent principalement aux premières entrées des modules réduits. Or, ceux-ci peuvent posséder un nombre quelconque d'entrées dans un ordre qui ne sera pas toujours le plus arrangeant pour les réduire. Une transformation permettant de changer l'ordre des entrées est donc requise.

Le module M prend en entrée deux valeurs in_1 et in_2 de types respectifs x et y, et renvoie une sortie *out* de type z. Le combinateur C permet de changer l'ordre des entrées.

$$Min_1in_2 \equiv CMin_2in_1$$

Ce qui permet de déduire la règle suivante :

$$\frac{[\mathrm{M} : \mathrm{FxFyz}]}{[(\mathrm{CM}) : \mathrm{FyFxz}]}\ \mathrm{C}$$

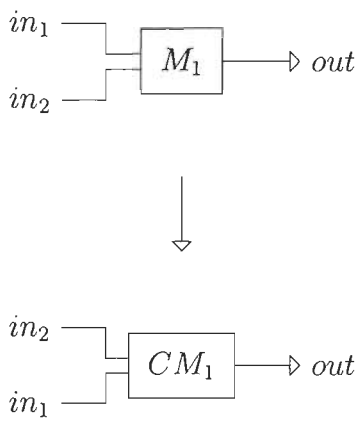


FIGURE 3.8 – Application de la règle de permutation

Avec cette règle, la permutation des deux premières entrées est possible (voir figure 3.8). Si cet échange peut régler certaines situations, on veut pouvoir réorganiser l'ordre des entrées selon nos besoins.

exemple Soit un module M_2 ayant quatre entrées in_1, in_2, in_3 et in_4 de types respectifs x_1,x_2,x_3 t x_4. Il est intégré dans une chaîne de traitement de telle sorte que $in_1 = in_4$. Ainsi, il ne peut pas être plus réduit, mais si l'entrée in_4 était en deuxième position, la règle de duplication serait applicable. Pour ce faire, deux permutations sont appliquées : la première entre in_4 et in_3, et la seconde entre in_4 et in_2 (voir Figure 3.9).

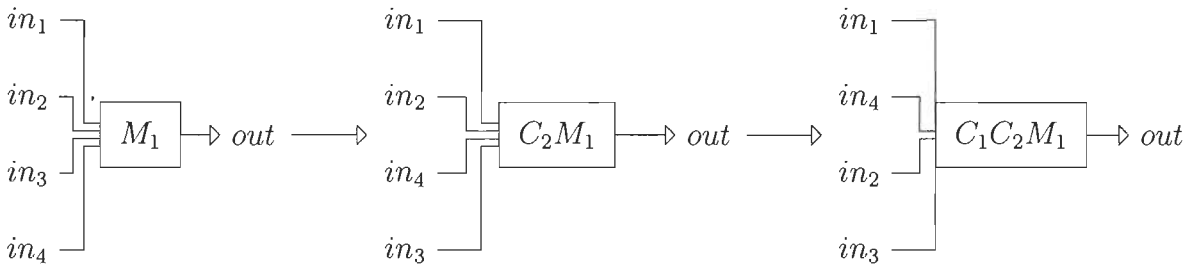


FIGURE 3.9 – Exemple d'utilisation de la règle de permutation

Ces déplacements peuvent se faire d'une position quelconque vers une autre. Pour notre utilisation, nous voudrons déplacer les entrées vers les premières places. La règle sera donc exprimée pour rapprocher l'entrée de la première et non l'inverse.

Généralisation

Le module M' a n entrées notées in_k de type x_k pour $k \in [|1, n|]$, et une sortie de type y. Soit $j \in [|1, n-1|]$. Pour échanger les entrées j et $j+1$, le combinateur C peut être appliqué à distance. C_0 échange les entrées 1 et 2, C_1 échange les entrées 2 et 3, et ainsi C_{j-1} échange les entrées j et $j+1$.

$$M' in_1 .... in_n \equiv C_{j-1} M' in_1 ... in_{j+1} in_j ... in_n$$

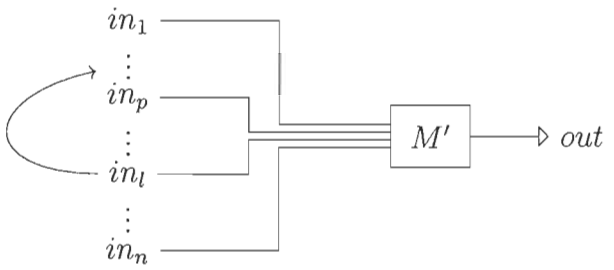


FIGURE 3.10 – Permutations successives

En effectuant des décallages successif, une entrée peut être déplacée d'une position quelconque l vers une autre position quelconque p, en gardant l'ordre des autres entrées (voir la Figure 3.10).

Ce décalage correspond à des permutations successives. Le nouveau module serait représenté par l'expression combinatoire $(C_{p-1}\,(C_p\,(\ldots\,(C_{l-2}M'))))$. Un déplacement de la position l vers la position p est représenté par la règle

Règle 5.

$$\frac{[M : Fx_1 \ldots Fx_n y]}{[(C_{p-1}\,(C_p\,(\ldots\,(C_{l-2}M)))) : Fx_1 \ldots Fx_{p-1} Fx_l Fx_p \ldots Fx_{l-1} Fx_{l+1} \ldots Fx_n y]}\ C$$

En reprenant l'exemple précédent : pour déplacer l'entrée de la position 4 à la position 2, $n = 4$, $l = 4$ et $p = 2$

$$\frac{[\mathrm{M} : Fx_1 Fx_2 Fx_3 Fx_4 y]}{[(C_{2-1}\,(C_2 M_1)) : Fx_1 Fx_4 Fx_2 Fx_3 y]}\ \mathrm{C}$$

3.3 Exemple de réduction d'une chaîne

La réduction d'une chaîne plus longue de modules se fait en appliquant les règles précédentes les unes après les autres, jusqu'à ne plus avoir qu'un seul module. L'exemple suivant illustre cette série de réductions.

Soient w,x,y et z des types et M_1, M_2, M_3, M_4, M_5, M_6 et M_7 des modules définis tel que suit :

$$M_1 : FxFyz$$
$$M_2 : Fzw$$
$$M_3 : Fzz$$
$$M_4 : FwFzx$$
$$M_5 : Fyw$$
$$M_6 : Fwx$$
$$M_7 : FxFxy$$

Sur ces sept modules, trois ont deux entrées (M_1, M_4 et M_7), alors que les autres n'en ont qu'une seule. Les types ont été choisis afin de pouvoir construire une chaîne d'exécution qui soit valide.
Un utilisateur souhaite les faire s'exécuter selon l'ordre représenté par la figure 3.11 et la chaîne 3.1.

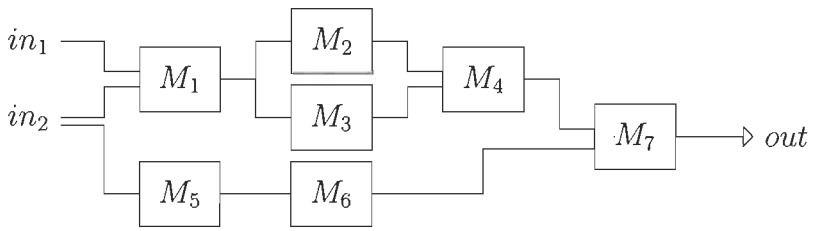


FIGURE 3.11 – Schéma de la chaine de module en exemple

$$out \equiv M_7\,(M_4\,(M_2\,(M_1 in_1 in_2))\,(M_3\,(M_1 in_1 in_2)))\,(M_6\,(M_5 in_2)) \tag{3.1}$$

Les règles définies précédemment doivent nous permettre d'obtenir ce même résultat. Pour les besoins de cet exemple, les règles seront appliquées du dernier module vers le premier, mais dans le chapitre suivant nous déterminerons l'ordre d'exécution et les algorithmes d'application qui ont été retenus. Cet ordre d'application nous amènera à

réduire totalement la première entrée du module M_7 puis à permuter ses entrées afin de traiter la seconde entrée.

Le premier module à regarder est donc le M_7. Il est précédé de deux modules (M_4 et M_6), la règle de composition peut être appliquée sur sa première entrée (voir Figure 3.12).

$$\frac{[M_4 : FwFzx] \quad + \quad [M_7 : FxFxy]}{[(B^2M_7M_4) : FwFzFxy]}\ B^2$$

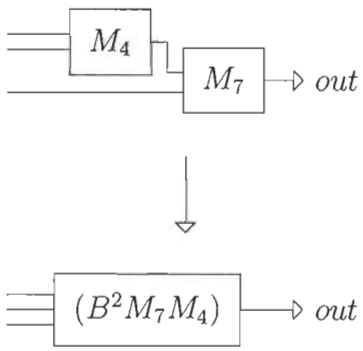


FIGURE 3.12 – Exemple de réduction - étape 1 - règle de composition

Le module courant est $(B^2M_7M_4)$. Il est précédé de trois modules (M_2, M_3, M_6), la règle de composition s'applique à sa première entrée (voir Figure 3.13).

$$\frac{[M_2 : Fzw] \quad + \quad [(B^2M_7M_4) : FwFzFxy]}{[(B(B^2M_7M_4)M_2) : FzFzFxy]}\ B$$

Le module courant est $(B(B^2M_7M_4)M_2)$. Il est précédé de trois modules (M_1, M_3 et M_6). La règle de composition ne peut pas être appliquée à sa première entrée car M_1 est également suivi de M_3. Appliquer la composition ferait disparaître le lien entre ces deux modules puisqu'il ne serait pas pris en compte et ferait disparaître M_1. Cependant, cette situation est celle de la règle de composition distributive. Celle-ci est donc appliquée (voir Figure 3.14).

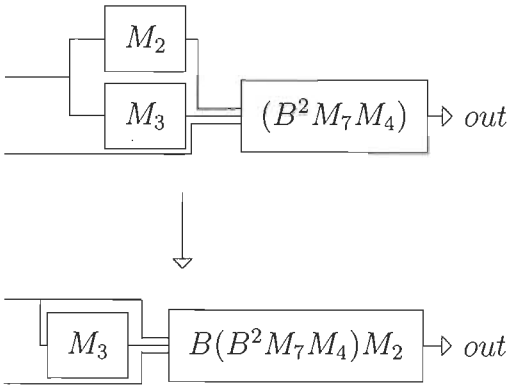


FIGURE 3.13 – Exemple de réduction - étape 2 - règle de composition

$$\frac{[M_3 : Fzz] \quad + \quad [(B(B^2M_7M_4)M_2) : FzFzFxy]}{[(S(B(B^2M_7M_4)M_2)M_3) : FzFxy]}\ S$$

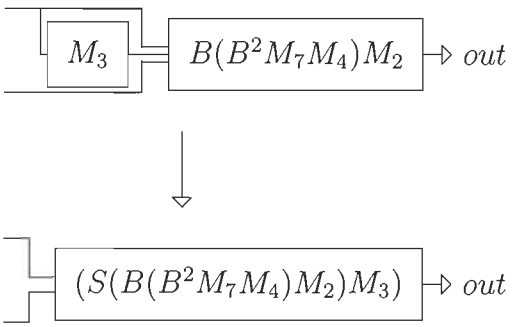


FIGURE 3.14 – Exemple de réduction - étape 3 - règle de composition distributive

Le module courant est $(S(B(B^2M_7M_4)M_2)M_3)$. Il est précédé de M_1 et de M_6. M_1 n'a plus que le module courant à sa suite et ainsi la règle de composition s'applique (voir Figure 3.15).

$$\frac{[M_1 : FxFyz] \quad + \quad [(S(B(B^2M_7M_4)M_2)M_3) : FzFxy]}{[(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1) : FxFyFxy]}\ B^2$$

Le module courant est $(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)$. La première entrée est totalement réduite. La chaîne obtenue est représentée par la figure 3.16.

Pour pouvoir réduire le reste de la chaîne, les entrées du module courant doivent être réorganisée. La dernière entrée doit être amenée à la première place afin que la

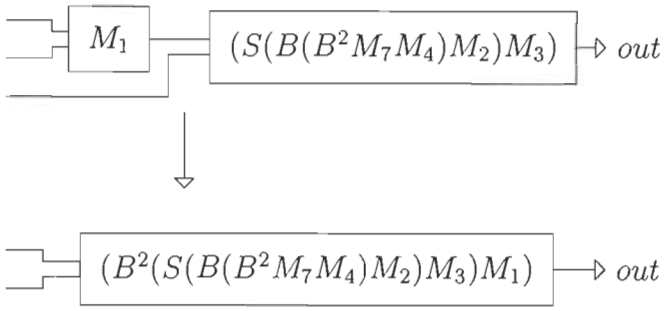


FIGURE 3.15 – Exemple de réduction - étape 4 - règle de composition

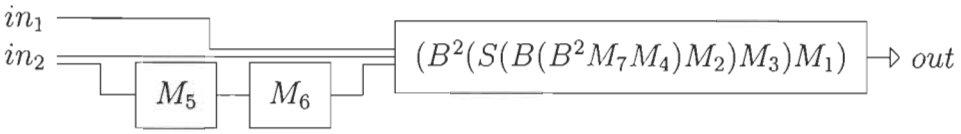


FIGURE 3.16 – Exemple de réduction - Réduction de la première entrée terminée

régle de permutation puisse s'appliquer (voir Figure 3.17).

$$\frac{[(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1) : FxFyFxy]}{[(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1))) : FxFxFyy]}\ C$$

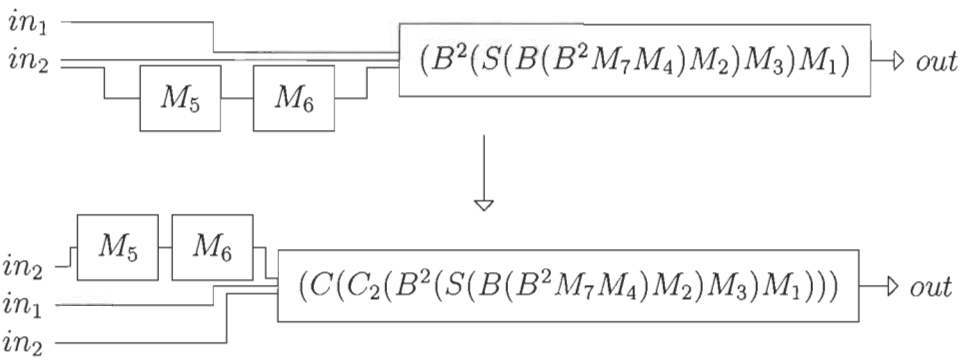


FIGURE 3.17 – Exemple de réduction - étape 5 - règle de permutation

La règle de composition peut ensuite être appliquée deux fois d'affilé. Une première fois pour réduire le module M_5 et la seconde pour le module M_6 (voir Figure 3.18).

$$\frac{[M_5 : Fyw] \qquad \dfrac{[M_6 : Fwx] \qquad [(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1))) : FxFxFyy]}{[(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6) : FwFxFyy]}\ B}{[(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5) : FyFxFyy]}\ B$$

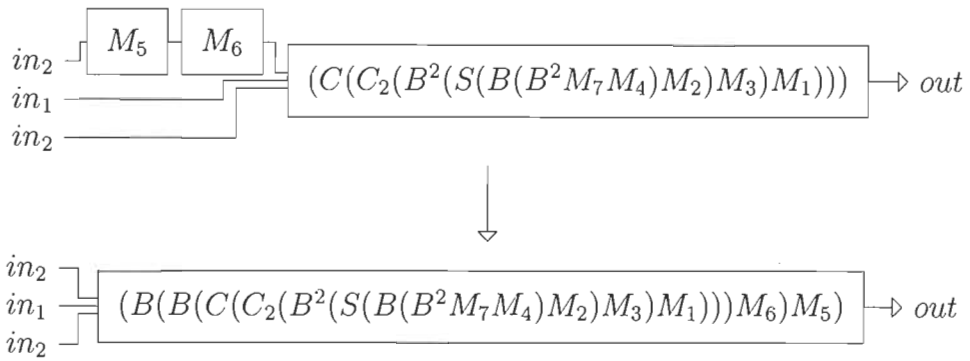


FIGURE 3.18 – Exemple de réduction - étape 6 et 7 - règle de composition

La chaîne est presque entièrement réduite puisqu'un seul module reste. Cependant, le module possède plusieurs fois la même entrée. En les rassemblant à l'aide de la règle de permutation, l'application de la règle de duplication pourra être effectuée (voir Figure 3.19).

$$\dfrac{\dfrac{[(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5) : FyFxFyy]}{[(C^2(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5)) : FyFyFxy]}\,C2}{[(W(C^2(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5))) : FyFxy]}\,W$$

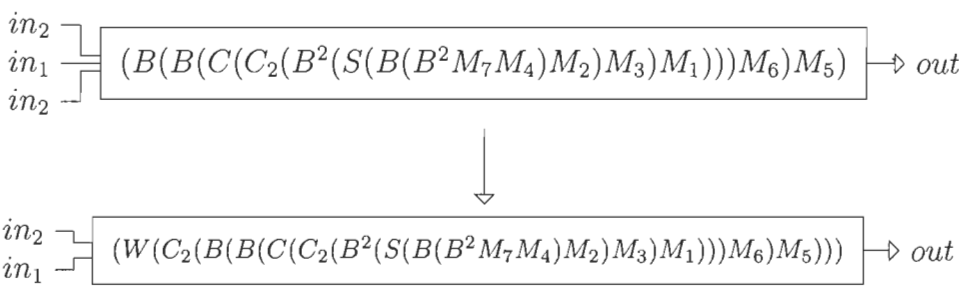


FIGURE 3.19 – Exemple de réduction - étape 8 et 9 - règle de permutation et de duplication

La chaîne est totalement réduite. L'agencement des modules les uns par rapport aux autres est donc représenté de manière complète par l'expression combinatoire $(W(C_2(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5)))$. En utilisant les

β-réductions des combinateurs la constituant, l'expression 3.1 devrait être obtenue.

$$
\begin{aligned}
&(W(C_2(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5)))in_2in_1\\
\equiv&(C_2(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5))in_2in_2in_1\\
\equiv&(B(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)M_5)in_2in_1in_2\\
\equiv&(B(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))M_6)(M_5in_2)in_1in_2\\
\equiv&(C(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)))(M_6(M_5in_2))in_1in_2\\
\equiv&(C_2(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1))in_1(M_6(M_5in_2))in_2\\
\equiv&(B^2(S(B(B^2M_7M_4)M_2)M_3)M_1)in_1in_2(M_6(M_5in_2))\\
\equiv&(S(B(B^2M_7M_4)M_2)M_3)(M_1in_1in_2)(M_6(M_5in_2))\\
\equiv&(B(B^2M_7M_4)M_2)(M_1in_1in_2)(M_3(M_1in_1in_2))(M_6(M_5in_2))\\
\equiv&(B^2M_7M_4)(M_2(M_1in_1in_2))(M_3(M_1in_1in_2))(M_6(M_5in_2))\\
\equiv&M_7(M_4(M_2(M_1in_1in_2))(M_3(M_1in_1in_2)))(M_6(M_5in_2))
\end{aligned}
$$

L'expression est bien identique. Les transformations ont donc laissé une trace de l'ordre d'exécution des modules. Aussi, en appliquant les règles, les types sont vérifiés, puisque les règles prennent en compte le type des modules. La chaine est donc bien construite et l'expression trouvée permet de connaître son ordre d'exécution.

3.4 Synthèse des règles

Ce chapitre a mis en place cinq règles de réduction et montré par un exemple une manière d'appliquer ces règles sur une chaîne de modules. Les règles qui ont été définies sont les règles d'application, de composition, de composition distributive, de duplication et de permutation. Elles sont définies comme suit :

(1) règle d'application

$$\frac{[X : x] \quad + \quad [M_1 : Fxy]}{[Y : y]}$$

(2) règle de composition

$$\frac{[M_1 : Fx_1 \ldots Fx_n y_1] \quad + \quad [M_2 : Fy_1 \ldots F_m z]}{[(B^n M_2 M_1) : Fx_1 \ldots Fx_n Fy_2 \ldots Fy_m z]}\ \mathrm{B}$$

(3) règle de composition distributive

$$\frac{[\mathrm{M1} : \mathrm{Fxy}] \quad + \quad [\mathrm{M2} : \mathrm{FxFyz}]}{[(\mathrm{SM2M1}) : \mathrm{Fxz}]}\ \mathrm{S}$$

(4) règle de duplication

$$\frac{[\mathrm{M} : (\mathrm{Fx...Fx})^{n\,fois}\mathrm{Fx}_{n+1} \ldots \mathrm{Fx}_m \mathrm{y}]}{[(\mathrm{W}^n\mathrm{M} : \mathrm{FxFx}_{n+1} \ldots \mathrm{Fx}_m \mathrm{y}]}\ \mathrm{W}$$

(5) règle de permutation

$$\frac{[\mathrm{M} : Fx_1 \ldots Fx_n y]}{[(C_{p-1}\,(C_p\,(\ldots(C_{l-2}M)))) : Fx_1 \ldots Fx_{p-1} Fx_l Fx_p \ldots Fx_{l-1} Fx_{l+1} \ldots Fx_n y]}\ \mathrm{C}$$

Chapitre 4

Implémentation

La théorie développée est composée de règles simples qui sont applicables par un ordinateur. Mais lors de leur application à une chaîne se présentent deux choix : quel est le module visé et quelle règle y appliquer. Pour ce faire, deux algorithmes ont été implémentés et testés. Chacun présente des avantages et des inconvénients.

4.1 Choix techniques

Le cœur de l'application a été conçu sous forme d'une librairie de fonctions. Une application permet d'appeler cette librairie pour en tester les fonctionnalités.

4.1.1 Librairie de fonctions

La logique utilisée pour traiter les chaînes de modules est fortement liée à la logique du λ-calcul, et donc à la même logique que celle utilisée par les langages du paradigme

fonctionnel. Le choix a donc été fait de prendre un langage fonctionnel pour mettre en place les règles et la réduction des chaînes. Pour permettre une communication simple avec d'autres langages et d'autres paradigmes, l'environnement de développement offert par Microsoft semblait le plus adapté. La librairie a donc été codée en F♯.

Elle est composée de trois modules.

— *MCObjects*

Définit ce qu'est un module et une chaîne de modules. Implémente les fonctions de manipulations utiles pour le reste de l'application, par exemple la récupération d'un module de la chaîne, la recherche du dernier module de la chaîne, l'ajout ou le retrait d'un module et la vérification de la validité de la chaîne.

— *MCRules*

Définit l'application des règles. Implémente également les fonctions de vérification qui permettent de savoir si une règle est applicable à un module de la chaîne.

— *MCReductions*

Définit les algorithmes de réduction des chaînes de modules. Détermine la logique d'application des règles qui sont définies dans le module précédent.

Lorsque la librairie est importée, seules les fonctions de MCReductions sont accessibles. Un programme externe n'aura pas besoin de plus que d'appeler les algorithmes de réduction.

4.1.2 Application de test

Pour s'assurer du fonctionnement de la librairie, celle-ci devait être testée sur des chaînes. Une application de test a donc été codée. Pour s'assurer de son utilisabilité avec un langage autre, le programme de test a été fait en C♯. Il permet d'appliquer les algorithmes implémentés sur des chaînes de module.

Chaînes de modules

À terme, les chaînes de modules seront mises en place par un utilisateur qui pourrait ainsi construire des chaînes de traitement et les tester à l'aide d'un logiciel qui utiliserait la bibliothèque implémentée. Pour les besoins des tests, les chaînes sont représentées dans un fichier XML. Celui-ci peut contenir plusieurs chaînes. Le nœud parent du fichier est `elements`. Il contient plusieurs chaînes représentées par le nœud `chain`. Chaque chaîne contient plusieurs modules représentés par le nœud `modul`. Chaque module est défini par :

- `id` l'id du module. Il sera utilisé pour définir les liens entre les modules.
- `types` les types des entrées du module et celui de sortie.
- `inputs` les entrées. Une valeur négative représente une entrée de la chaîne complète alors qu'une valeur positive est l'id du module qui précède cette entrée.
- `combchain` le nom qui représentera le module dans la chaîne combinatoire à la fin.

Par exemple, un module M_1 ayant deux entrées identiques de type entier et retournant une chaîne de caractères sera représenté par :

```
<modul>
  <id>1</id>
  <types>
    <type>int</type>
    <type>int</type>
    <type>string</type>
  </types>
  <inputs>
    <input>-1</input>
    <input>-1</input>
  </inputs>
  <combchain>M1</combchain>
</modul>
```

Un enchaînement de deux modules M_1 et M_2, tels que M_1 est précédé par M_2 et que chacun soit de type F_{IntInt}, sera représenté par :

```
<chain>
  <modul>
    <id>1</id>
    <types>
  <type>int</type>
  <type>int</type>
    </types>
    <inputs>
  <input>2</input>
    </inputs>
    <combchain>M1</combchain>
  </modul>
  <modul>
    <id>2</id>
    <types>
  <type>int</type>
  <type>int</type>
    </types>
    <inputs>
  <input>-1</input>
    </inputs>
    <combchain>M2</combchain>
  </modul>
</chain>
```

Ainsi le programme peut tester la réduction de chaînes de traitement, en prenant en compte les types d'entrées et de sortie des modules pour vérifier la validité de ces chaînes.

Utilisation de l'application

L'utilisation de l'application de test se fait à l'aide d'une interface ne ligne de commande. Pour rappel, ce type d'interface n'a été mise en place qu'à titre de test et un programme de construction de chaîne qui soit adaptée à des utilisateurs plus variés doit être mis en place dans le futur. Ce programme permet de tester la librairie sur des chaînes de traitement. Lors de son appel, il peut prendre plusieurs arguments :

- `file` (obligatoire) précise l'emplacement du fichier XML qui contient la définition des chaînes à réduire
- `method` (1 par défaut) précise la méthode de réduction à employer.(voir 4.2.2) 1 : Dès que possible 2 : Ascendante
- `time` (suivit du nombre d'itération) permet d'effectuer chaque réduction plusieurs fois et de calculer le temps. Permet de comparer les algorithmes.
- `silent` (faux par défaut) permet de prévenir l'affichage du résultat. Vrai par défaut lorsqu'on active l'option `time`.
- `verbose` (faux par défaut) permet d'afficher les détails de la réduction.

Pour le premier ensemble de test, qui sera détaillé dans la section 4.3, le programme rend le résultat suivant :

```
(WM1)
(BM1M2)
(SM1M2)
```

Si l'option `verbose` est activée, le résultat pour la première chaîne de cet ensemble sera le suivant :

```
Verbose
Modules de départ : "M1"
Boucle principale
tests de duplication
On peut appliquer la règle de duplication
M1 -> (WM1)
```

```
règle de duplication appliquée
 liste active : "(WM1)"
tests de duplication
test de composition
test de composition àdistance n
test de composition distributive
Boucle principale
tests de duplication
test de composition
test de composition àdistance n
test de composition distributive
test de règle de permutation
liste active"(WM1)"
(WM1)
```

Pour cet exemple, l'algorithme par défaut est utilisé.

Ce programme permet de tester les algorithmes implémentés sur diverses chaînes de traitement, de connaître leur temps d'exécution, et d'avoir une trace de leur exécution pour vérifier qu'ils agissent bien comme attendu.

4.2 Algorithme d'application

Pour concevoir l'algorithme d'application des règles de réductions, il faut répondre à deux questions. La première est de savoir s'il existe des situations dans lesquelles plusieurs règles sont applicables, et le cas échéant quelle règle aura la priorité. La seconde concerne l'ordre de traitement des modules de la chaîne.

4.2.1 Priorité d'application

Avant de déterminer la priorité d'application des règles, il nous faut savoir s'il arrive que plusieurs règles soient applicables. La règle applicative est utile lors de l'exécution de la chaîne, mais ne servira pas à la réduire. La règle de permutation est applicable sur tous les modules ayant plusieurs entrées, mais sera utilisée uniquement pour réorganiser les entrées dans le cas où aucune autre règle ne s'applique. Il reste trois règles à vérifier.

— Duplication

— Composition

— Composition distributive

Pour que les règles de duplication et de composition soient applicables, il faut que les deux premières entrées d'un module M' soient les mêmes et que la première soit précédée d'un module M. Ce qui correspond à la situation illustrée par le schéma 4.1. Si la règle de composition est appliquée en première, la réduction est $S(BM'M)M$. Si la règle de duplication est appliquée en première, la réduction est $B(WM')M$.

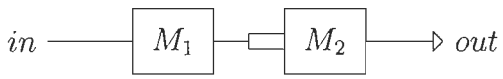


FIGURE 4.1 – Cas d'ambiguïté entre la composition et la duplication

Pour que les règles de composition et de composition distributive soient applicables il faut que la première entrée d'un module M_3 soit précédée d'un module M_1 et que la seconde soit précédée d'un module M_2, lui-même précédé de M_1. Cette situation est illustrée pas schéma 4.2. Si la règle de composition est appliquée en première, la réduction est $B(B(C(BM_3M_1))M_2)M_1$. Si la règle de composition distributive est appliquée en première, la réduction est $B(SM_3M_2)M_1$.

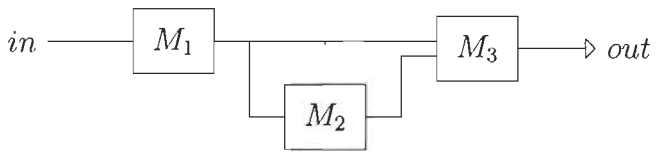


FIGURE 4.2 – Cas d'ambiguïté entre la composition et la composition distributive

Aucun cas d'ambiguïté ne se pose entre la duplication et la composition distributive.

Deux cas d'ambiguïté se posent. Dans les deux cas, le fait d'appliquer la règle de composition en première duplique l'un des modules présents et allonge la chaîne réduite. Aussi, nous avons voulu éviter la duplication des modules afin d'alléger le programme.

Cette situation apparaît dans trois cas. Le premier est celui représenté au schéma 3.5 (page 27), où l'application de la règle de composition au module M_4 permet d'éviter le problème. Le second est représenté par le schéma 4.2, où l'application de la règle de composition distributive permet de l'éviter aussi. Le dernier est représenté par le schéma 4.1, où l'application de la règle de duplication distributive l'évite également.

La décision a donc été prise de refuser l'application de la règle de composition lorsqu'elle entraîne une duplication du module précédent. De la même manière et pour la même raison, la règle de composition distributive n'est pas appliquée si le module précédent la deuxième entrée doit être dupliqué pour cela.

En enlevant ces cas particuliers, les cas d'ambiguïté n'existent plus et les règles peuvent être testées dans n'importe quel ordre sans qu'il n'y ait de conséquences puisqu'une seule est applicable à chaque fois.

4.2.2 Deux approches choisies

Si, lorsqu'une réduction est appliquée, l'ordre de priorité est établi entre les différentes règles applicables, il faut déterminer sur quel module les règles seront testées et appliquées. Des algorithmes ont été implémentés et testés. D'autres pourraient être mis en place et testés dans des travaux futurs. Les deux algorithmes choisis l'ont été pour

illustrer d'une part une approche centrée sur les modules, c'est-à-dire que pour un module choisi les différentes règles sont testées, et d'autre part une approche centrée sur les règles, c'est-à-dire que pour une règle choisie, les différents modules de la chaîne sont testés.

Application ascendante

Dans l'exemple du chapitre 3, la méthode utilisée est ascendante. Le premier module réduit est le dernier de la chaîne. Au fur et à mesure des opérations, la chaîne est de plus en plus courte jusqu'à ne contenir qu'un seul module. Lorsqu'aucune règle ne s'applique, la réduction de termine. S'il y a encore plus d'un seul module dans la chaîne, un message d'erreur est retourné. Sinon la réduction est considérée comme réussie. Ce fonctionnement est représenté par l'algorithme 4.1.

Cette méthode est centrée sur les modules. En effet, ils sont traités un par un, et chaque règle est testée à chaque passage dans la boucle principale. Lorsqu'une situation non résolvable apparaît dans la chaîne, un résultat incomplet sera rendu. La linéarité du traitement laissera tous les modules du début de la chaîne non traités. Ainsi, en cas d'erreurs, seule celle qui se trouve le plus proche de la fin de la chaîne dans l'ordre de traitement de celle-ci sera visible. Celles qui pourraient être présentes dans le début de la chaîne restent non traitées.

Les algorithmes de test d'applicabilité des règles sont de complexité $o(n)$ avec n le nombre de modules, car ils demandent de parcourir l'ensemble des modules pour vérifier que rien n'empêche la réduction voulue. Dans cette approche ascendante, chaque passage correspond à l'application d'une règle. Seule la règle de permutation n'entraîne pas de réduction du nombre de modules dans la chaîne. La boucle principale sera donc parcourue un nombre de fois proportionnel à n. Ainsi la complexité de l'algorithme est en $o(n)$.

Algorithm 4.1 Algorithme ascendant

```
estReduit ← vrai
moduleCourant ← dernierModule
while estReduit do
    estReduit ← faux
    if peutAppliquerDuplication(moduleCourant) then
        appliquerDuplication(moduleCourant)
        estReduit ← vrai
    else
        if peutAppliquerComposition(moduleCourant) then
            appliquerComposition(moduleCourant)
            estReduit ← vrai
        else
            if peutAppliquerCompositionDistributive(moduleCourant) then
                appliquerCompositionDistributive(moduleCourant)
                estReduit ← vrai
            else
                moduleTemporaire ← moduleCourant
                trierEntrees(moduleCourant)
                if moduleTemporaire ≠ moduleCourant then
                    estReduit ← vrai
                end if
            end if
        end if
    end if
end while
```

Exemple d'application :

Soit la chaîne de modules de la figure 4.3.

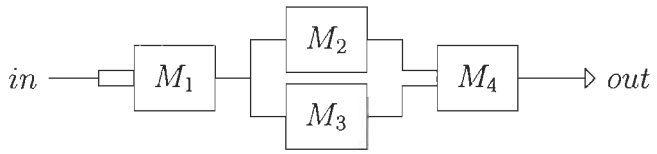


FIGURE 4.3 – Exemple d'application - Chaîne de départ

Au début de l'exécution, le module courant est initialisé au dernier module de la chaîne. Ici, le module M_4. Les règles sont testées chacune leur tour :

— Duplication : *non applicable*

— Composition : *applicable*

La règle de composition est donc appliquée au module courant (voir Figure 4.4).

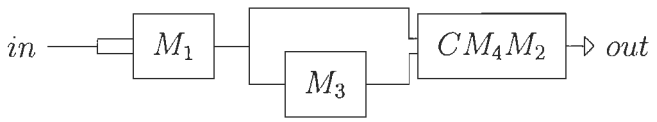


FIGURE 4.4 – Exemple d'application - approche ascendante - réduction 1

Le module ainsi réduit devient le module courant : CM_4M_2. Les règles sont testées à nouveau :

— Duplication : *non applicable*

— Composition : *non applicable*

— Composition distributive : *applicable*

La règle de composition distributive est donc appliquée (voir Figure 4.5).

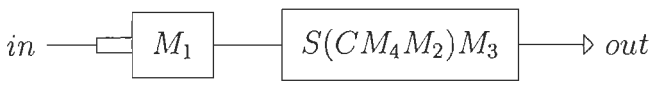


FIGURE 4.5 – Exemple d'application - approche ascendante - réduction 2

Le module réduit devient le module courant : $S(CM_4M_2)M_3$. Les règles sont testées.

— Duplication : *non applicable*

— Composition : *applicable*

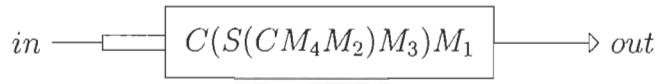


FIGURE 4.6 – Exemple d'application - approche ascendante - réduction 3

La règle de composition est appliquée (voir Figure 4.6).

Le module réduite devient le module courant : $C(S(CM_4M_2)M_3)M_1$. Les règles sont testées.

— Duplication : *applicable*

La règle de duplication est appliquée (voir Figure 4.7).

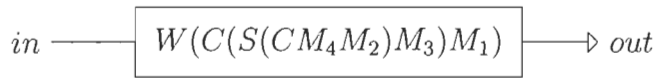


FIGURE 4.7 – Exemple d'application - approche ascendante - réduction 4

Le module réduit devient le module courant : $W(C(S(CM_4M_2)M_3)M_1)$. Les règles sont testées.

— Duplication : *non applicable*

— Composition : *non applicable*

— Composition distributive : *non applicable*

— Tri des entrées : *non applicable*

Aucune règle n'est applicable, on quitte la boucle principale et vérifie si le module courant est le dernier module restant. C'est le cas, la chaîne de module a donc été réduite complètement et le module courant est renvoyé comme résultat.

Dès que possible

Une alternative afin de pallier au désavantage de l'algorithme précédent est de réduire la chaîne partout à la fois, selon les possibilités qui s'offrent. Cette stratégie se traduit par l'algorithme 4.2

Algorithm 4.2 Algorithme 'Dès que possible'

```
reduitGrandeBoucle ← vrai
while reduitGrandeBoucle do
    reduitGrandeBoucle ← faux
    reduitPetiteBoucle ← vrai
    while reduitPetiteBoucle do
        reduitPetiteBoucle ← faux
        for all Module m dans chaine do
            if peutAppliquerDuplication(m) then
                appliquerDuplication(m)
                reduitGrandeBoucle ← vrai
                reduitPetiteBoucle ← vrai
            end if
        end for
    end while
    reduitPetiteBoucle ← vrai
    while reduitPetiteBoucle do
        reduitPetiteBoucle ← faux
        for all Module m dans chaine do
            if peutAppliquerComposition(m) then
                appliquerComposition(m)
                reduitGrandeBoucle ← vrai
                reduitPetiteBoucle ← vrai
            end if
        end for
    end while
    reduitPetiteBoucle ← vrai
    while reduitPetiteBoucle do
        reduitPetiteBoucle ← faux
        for all Module m dans chaine do
            if peutAppliquerCompositionDistributive(m) then
                appliquerCompositionDistributive(m)
                reduitGrandeBoucle ← vrai
                reduitPetiteBoucle ← vrai
            end if
        end for
    end while
    if reduitGrandeBoucle = faux then
        for all Module m dans chaine do
            trierEntrees(m)
            if m a changé then
                reduitGrandeBoucle ← vrai
            end if
        end for
    end if
end while
```

Cette méthode est centrée sur les règles. Plutôt que de tester les règles pour un module donné, il teste tous les modules de la chaîne pour une règle donnée. La réduction se fait donc au fur et à mesure, sur toute la chaîne en même temps. Son avantage par rapport à l'approche précédente, est qu'en cas d'erreur, si la réduction ne peut pas être complétée, les sous-chaînes réductibles sont réduites.

Comme pour l'algorithme précédent, le nombre de passages dans la boucle principale est proportionnel au nombre de modules dans la chaîne. Aussi, puisque pour chaque règle est testée sur chaque module, et que le test a une complexité en $o(n)$, il y a 3 boucles imbriquées, qui mènent à une complexité en $o(n)$. Cet algorithme sera donc plus long à l'exécution que le précédent, ce qui est confirmé par les résultats de la section suivante.

Exemple d'application

Soit la même chaîne pour l'exemple de l'algorithme précédent (figure 4.3) La première règle testée est celle de duplication :

— M_1 : *applicable*

La règle de buplication est appliquée au module M_1 (voir Figure 4.8).

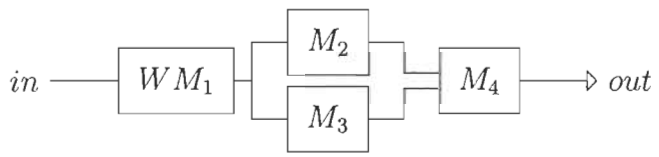


FIGURE 4.8 – Exemple d'application - dès que possible - réduction 1

Les tests continuent pour les modules suivants :

— M_2 : *non applicable*

— M_3 : *non applicable*

— M_4 : *non applicable*

Puisqu'une réduction a eut lieu, durant ce premier passage, un deuxième est effectué.

— WM_1 : *non applicable*

— M_2 : *non applicable*

— M_3 : *non applicable*

— M_4 : *non applicable*

Aucune réduction n'a été effectuée. C'est au tour de la règle de composition d'être testée sur les modules :

— WM_1 : *non applicable*

— M_2 : *applicable*

La règle de composition est appliquée au module M_2 (voir Figure 4.9).

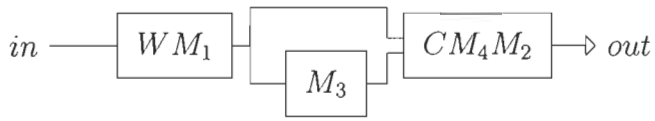


FIGURE 4.9 – Exemple d'application - approche dès que possible - réduction 2

Les tests continuent pour les modules suivants :

— M_3 : *non applicable*

Puisqu'il y a eut une réduction, les modules sont testés à nouveau.

— WM_1 : *non applicable*

— M_3 : *non applicable*

— CM_4M_2 : *non applicable*

La règle suivante à tester est celle de composition distributive :

— WM_1 : *non applicable*

— M_3 : *non applicable*

— CM_4M_2 : *applicable*

La règle de composition est appliquée au module CM_4M_2 (voir Figure 4.10).

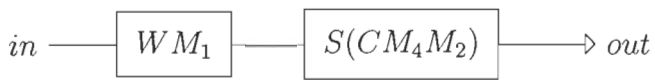


FIGURE 4.10 – Exemple d'application - approche dès que possible - réduction 3

Puisqu'il y a eut une réduction, les modules sont testés à nouveau :

— WM_1 : *non applicable*

— $S(CM_4M_2)$: *non applicable*

Puisque des réduction ont eut lieu, la règle de permutation n'est pas testée. En effet, elle n'est utilisée qu'en dernier recours. C'est donc la règle de duplication qui est testée à nouveau.

— WM_1 : *non applicable*

— $S(CM_4M_2)$: *non applicable*

Aucune application possible, la règle de composition est testée à son tour :

— WM_1 : *non applicable*

— $S(CM_4M_2)$: *applicable*

La règle est donc appliquée au module $S(CM_4M_2)$ (voir Figure 4.11).

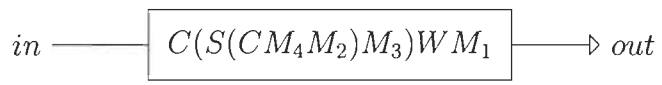


FIGURE 4.11 – Exemple d'application - approche dès que possible - réduction 4

Puisqu'il y a eut une réduction, la règle de composition est testée à nouveau :

— $C(S(CM_4M_2)M_3)WM_1$: *non applicable*

Aucune réduction n'a eut lieu. La règle de composition distributive est testée :

— $C(S(CM_4M_2)M_3)WM_1$: *non applicable*

Puisqu'une réduction aeut lieu durant cette boucle, on retourne à la règle de duplication :

— $C(S(CM_4M_2)M_3)WM_1$: *non applicable*

Puis à celle de composition :

— $C(S(CM_4M_2)M_3)WM_1$: *non applicable*

De composition distributive :

— $C(S(CM_4M_2)M_3)WM_1$: *non applicable*

Puisqu'aucune réduction n'a été faire, la règle de permutation est testée également :

— $C(S(CM_4M_2)M_3)WM_1$: *non applicable*

La réduction se termine, et puisqu'il ne reste qu'un seul module, la réduction est réussie et le module restant est renvoyé.

4.3 Tests et résultats

Afin de tester les algorithmes et de les comparer, un ensemble de chaînes de test a été mis en place. D'une part des chaînes valides et d'autre part des chaînes comprenant des erreurs.

Chaînes de test

Les premiers tests ont été faits sur des chaînes courtes. Celles-ci représentent l'application d'une seule règle (voir figure 4.12). Ces chaînes ont permis de prouver que les règles étaient bien implémentées.

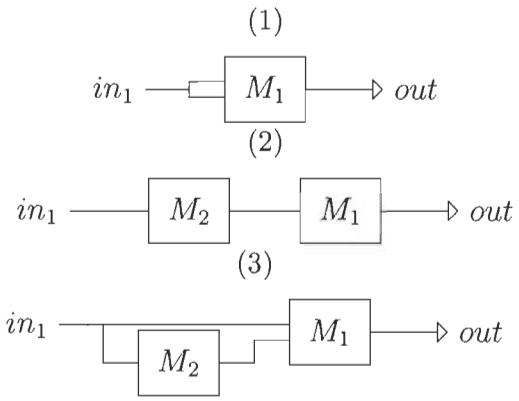


FIGURE 4.12 – Ensemble de test 1

Une deuxième série de tests, constituée de chaînes demandant l'application de plusieurs règles a ensuite été testée (voir 4.13). Le but était de faire appliquer plusieurs réductions d'affilée avant de passer à des chaînes plus longues. On a tenté de représenter avec ces deux séries, le plus de situations possible. En effet, on peut créer une infinité de chaînes à partir de ces chaînes en remplaçant un module de cette chaîne par une chaîne réductible. Par récursivité, ces nouvelles chaînes créées seront réductibles.

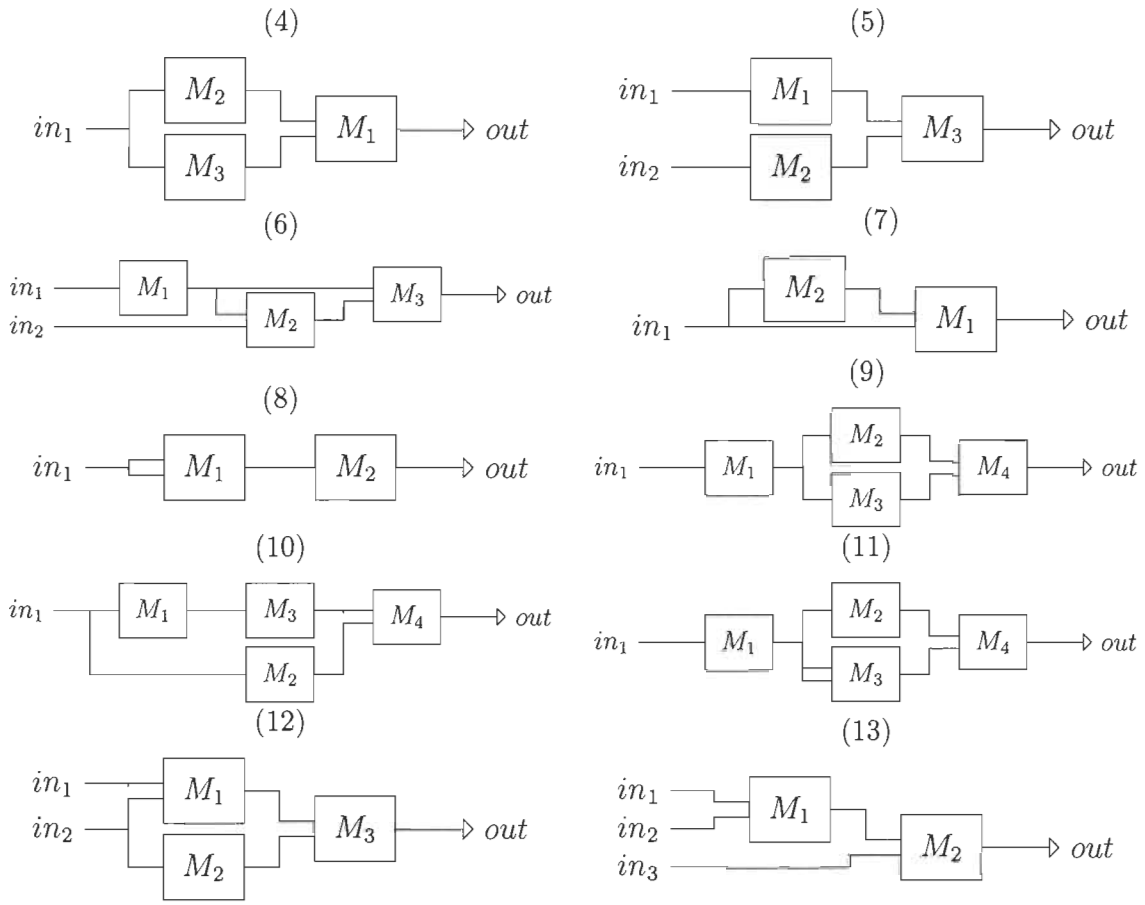


FIGURE 4.13 – Ensemble de test 2

Aussi, pour déterminer si les algorithmes sont capables de reconnaître une chaîne bien construite d'une chaîne comportant des erreurs, ils ont également été appliqués à des chaînes invalides, présentées par la figure 4.14. Afin de déterminer si une chaîne est valide, plusieurs points sont vérifiés :

- si plusieurs modules ont pour entrée la même entrée de la chaîne, le type doit correspondre
- si un module a pour entrée la sortie d'un autre module, les types doivent correspondre
- un module de la chaîne est le dernier module, c'est-à-dire qu'aucun autre module ne prend sa sortie pour entrée, et il est unique

Les deux approches ont été testées sur ces trois ensembles de chaînes et peuvent ainsi être comparées. Pour tester de manière plus poussée les algorithmes proposés, un autre ensemble de tests a été créé. Il est constitué de chaînes un peu plus longues

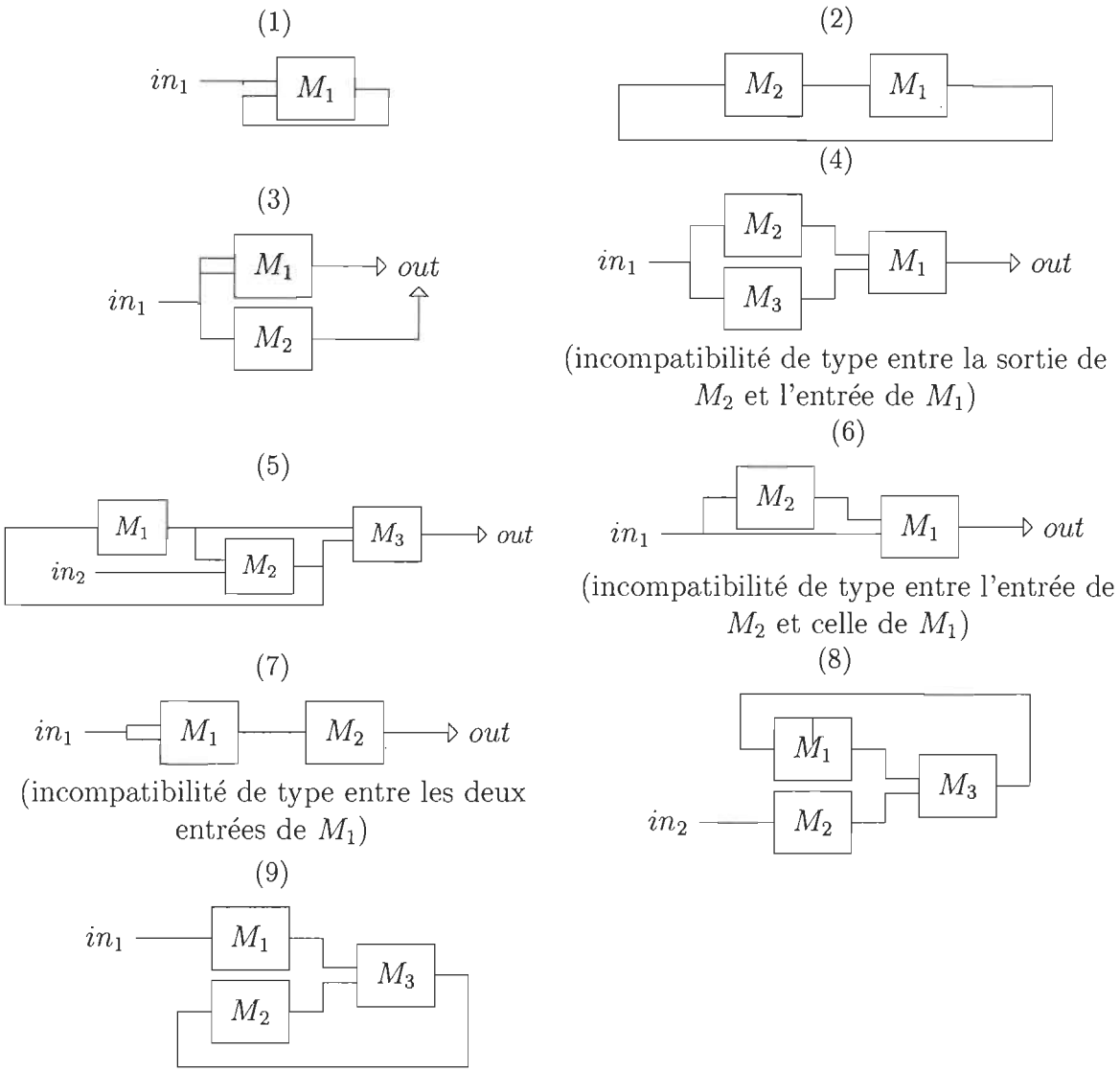


FIGURE 4.14 – Ensemble de test invalide

et tente de représenter des situations plus complexes. Toutes ces chaînes sont visibles dans l'annexe Annexe A Test des algorithmes.

Résultats

Les 13 chaînes valides et 9 chaînes non valides testées offrent un aperçu des performances des algorithmes. Le tableau 4.1 présente la validité des résultats pour le premier algorithme (ascendant), et le tableau 4.2 pour le second (dès que possible). Pour les chaînes testées, aucune chaîne non valide n'a été retournée comme valide et

toutes les chaînes valides ont été traitées avec succès.

	Non Valide	Valide
Réduite	0	13
Non réduite	9	0

TABLEAU 4.1 – Résultat de l'exécution de l'algorithme ascendant

	Non Valide	Valide
Réduite	0	13
Non réduite	9	0

TABLEAU 4.2 – Résultat de l'exécution de l'algorithme 'dès que possible'

Une option pour déterminer les temps de calcul a été mise en place pour le programme de test. Les résultats d'exécution sont présentés par le tableau 4.3. L'algorithme systématique y est clairement supérieur en terme de performance. Ce qui est en accord avec les complexités algorithmiques calculées précédemment.

test	(1)	(2)	(3)	(4)	(5)	(6)
Dès que possible	11	19	18	31	62	107
Systématique	4	7	6	12	17	24

TABLEAU 4.3 – Temps d'exécution des algorithmes de réduction en ms pour 500 passages

Les deux algorithmes permettent de réduire les chaînes testées et, par récursivité, l'ensemble des chaînes obtenues en remplaçant une de ces chaînes par une autre de ces chaînes. L'absence de cas d'erreurs pour les chaînes testées permet d'envisager l'utilisation de la bibliothèque implémentée dans le cadre d'un programme qui permettrait à un utilisateur, par une interface, de construire lui même ses chaînes de traitement et de les exécuter.

Conclusion

Lors du développement de logiciels, une grande partie du temps est perdue à coder des choses qui ont déjà été faites. La variété des langages et des paradigmes, ainsi que le désir et le besoin de garder les outils développés internes afin de ne pas perdre le fruit du travail fourni, fait que les outils existants sont souvent inutilisables ou inaccessibles. C'est dans le but de proposer une piste de solution à cette problématique que ce travail a été mené.

Les chaînes de traitement sont un outil permettant de faire travailler ensemble des programmes qui n'auraient pas pu collaborer naturellement. Elles se composent de programmes indépendants, possédant chacun une ou plusieurs entrées et une sortie. Ceux-ci s'enchaînent alors en ne communiquant que par une valeur qui est passée de l'un à l'autre. En utilisant les systèmes applicatifs et la logique combinatoire, un modèle théorique a été monté. Il se compose de règles de réduction qui permettent de déterminer si une chaîne de traitement est bien montée et de la représenter sous forme d'une application et d'une chaîne combinatoire. Cette théorie a été publiée en 2013 [3].

Deux méthodes ont été implémentées et testées. Pour ce faire, des chaînes de modules ont été conçues dans le but de représenter le plus de cas possible. Ces tests ont fait l'objet d'une seconde publication en 2015 [4]. Toutes les chaînes ont été résolues par les méthodes implémentées, ce qui signifie que toute chaîne de module qui serait

composée à partir de ces chaînes peut être réduite par l'application. Aussi, des chaînes non valides ont été passées au système et le système ne les résout pas.

Notre théorie a été montée et démontrée, il serait à présent intéressant d'y ajouter une interface complète pour que des utilisateurs puissent l'utiliser. Aussi, d'autres stratégies d'application des règles pourraient être implémentées.

Bibliographie

[1] *Apache UIMA - Apache UIMA*. URL : https://uima.apache.org/ (visité le 30/06/2015).

[2] Doris APPLEBY. *Programming Languages : Paradigm and Practice*. New York : Mcgraw-Hill College, jan. 1991.

[3] Ismaïl BISKRI, Marie ANASTACIO, Adam JOLY et Boucif AMAR BENSABER. "Integration of Sequence of Computational Modules Dedicated to Text Analysis : a Combinatory Typed Approach". (2013).

[4] Ismaïl BISKRI, Marie ANASTACIO, Adam JOLY et Boucif AMAR BENSABER. "A typed applicative system for a language and text processing engineering". *Journal of Innovation in Digital Ecosystems* (2015).

[5] Ismaïl BISKRI et Jean-Pierre DESCLÉS. "Applicative and Combinatory Categorial Grammar (from syntax to functional semantics)". *Recent Advances in Natural Language Processing*. John Benjamins Publishing Company. Ruslan Mitkov, Nicolas Nicolov, 1997.

[6] James COOKE BROWN. "Loglan". *Scientific American* (1960).

[7] H CUNNINGHAM, D MAYNARD, K BONTCHEVA et V TABLAN. "GATE : A framework and graphical development environment for robust NLP tools and applications". Proceedings of the 40th Annual Meeting of the ACL. 2002. (Visité le 30/06/2015).

[8] Haskell Brooks CURRY et Robert FEYS. *Combinatory Logic*. North-Holland Publishing Company, 1958. 446 p.

[9] *D2K - ToolCenter.* URL : http://echo.gmu.edu/toolcenter-wiki/index.php?title=D2K (visité le 11/05/2015).

[10] J. Stephen DOWNIE, John UNSWORTH, Bei YU, David TCHENG, Geoffrey ROCKWELL et Stephen J. RAMSAY. "A revolutionary approach to humanities computing ? : Tools development and the D2K data-mining framework". *Annual Joint Conference of The Association for Computers and the Humanities & The Association for Literary and Linguistic Computing.* 2005.

[11] *GATE.ac.uk - index.html.* URL : https://gate.ac.uk/ (visité le 11/05/2015).

[12] J. Roger HINDLEY, B. LERCHER et J. P. SELDIN. *Introduction to Combinatory Logic.* CUP Archive, 1972.

[13] Paul HUDAK, John HUGHES, Simon PEYTON JONES et Philip WADLER. "A history of Haskell : being lazy with class". *Proceedings of the third ACM SIGPLAN conference on History of programming languages.* ACM, 2007, p. 12–1.

[14] *KNIME — Open for Innovation.* URL : https://www.knime.org/ (visité le 11/05/2015).

[15] *Le grand dictionnaire terminologique.* URL : http://www.gdt.oqlf.gouv.qc.ca/ (visité le 15/08/2014).

[16] *Orange Data Mining.* URL : http://orange.biolab.si/ (visité le 11/05/2015).

[17] *RapidMiner.* URL : https://rapidminer.com/ (visité le 11/05/2015).

[18] Marc-André ROCHETTE. "Vers une ingénierie flexible pour le traitement de l'information". masters. Trois-Rivières : Université du Québec à Trois-Rivières, 2008.

[19] M. SCHÖNFINKEL. "Über die Bausteine der mathematischen Logik". *Math. Ann.* 92.3 (sept. 1924), p. 305–316.

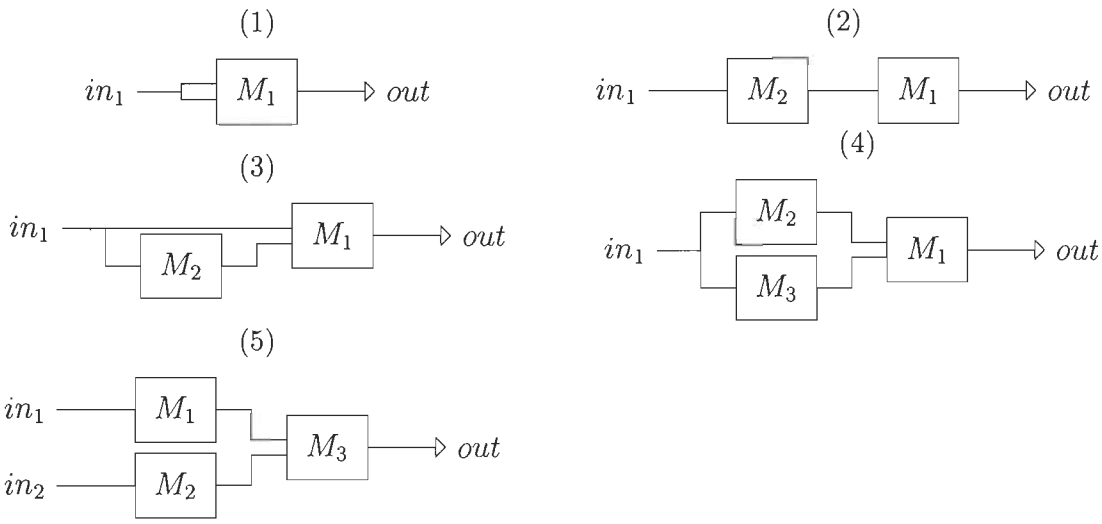
[20] S. K. SHAUMYAN. "Two Paradigms Of Linguistics : The Semiotic Versus Non-Semiotic Paradigm". *Web Journal of Formal, Computational and Cognitive Linguistics* (1998).

[21] A.W. WARR. "Integration, Analysis and Collaboration. An Update on Workflow and Pipelining in Cheminformatics". (2007).

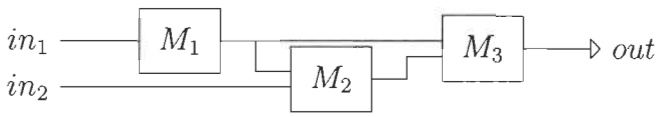
Annexe A Test des algorithmes

Annexe A.1 Chaînes de test valides

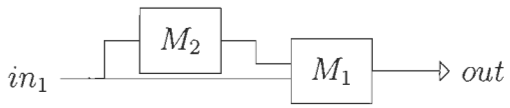
Les algorithmes ont été testés sur un ensemble de 23 chaînes de traitement bien construites. Les trois premières demandent l'application d'une seule règle et ont servit à valider l'implémentation des règles énoncées. Les chaînes (4) à (13) ont permis de valider les algorithmes sur des situations n'impliquant que deux à quatres modules, avant de leur donner des chaînes plus longues. Enfin, les chaînes (14) à (23) son composées d'un plus grand nombre de module et ont été composées afin de tester plus de cas possibles.



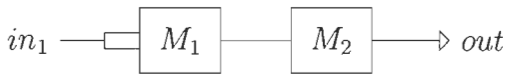
(6)



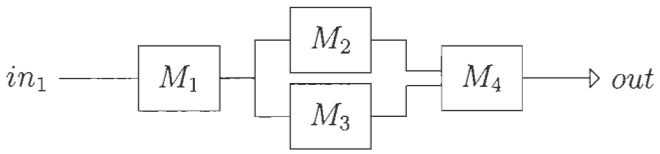
(7)



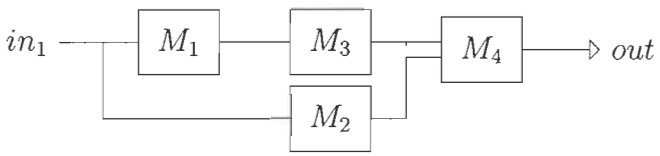
(8)



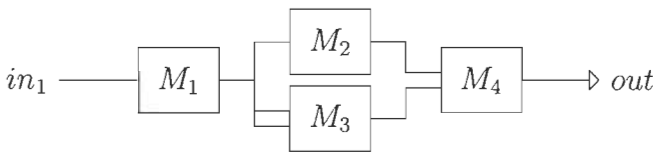
(9)



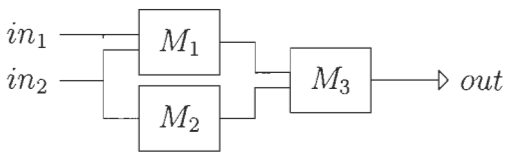
(10)



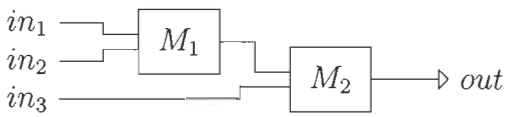
(11)



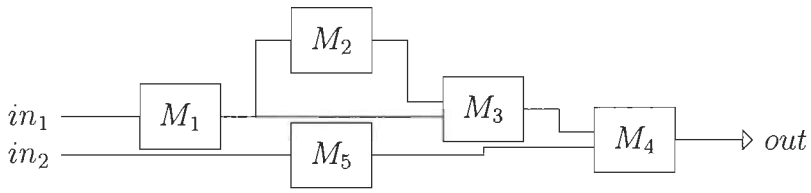
(12)



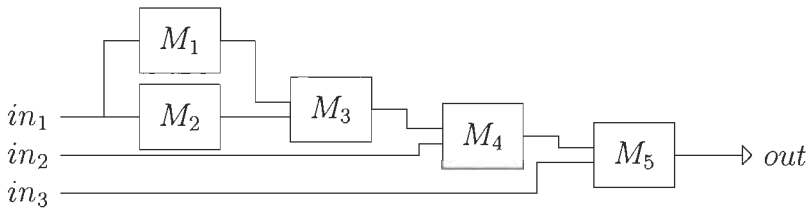
(13)



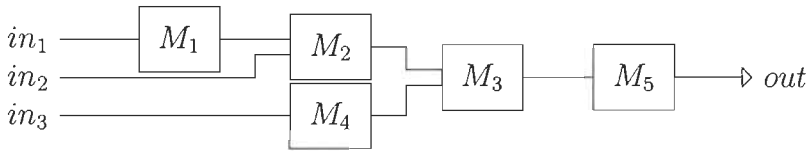
(14)



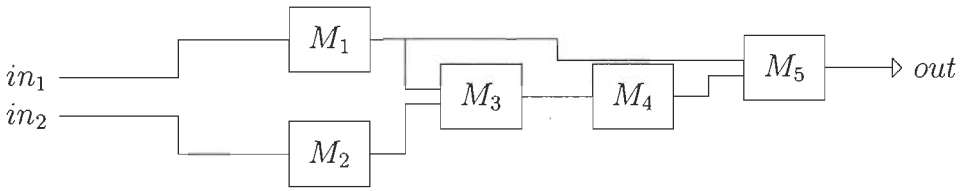
(15)



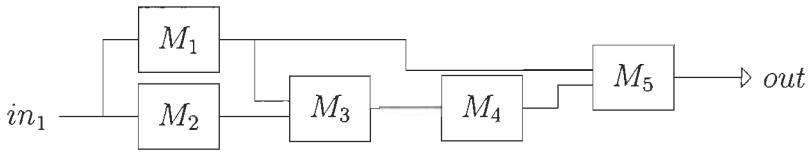
(16)



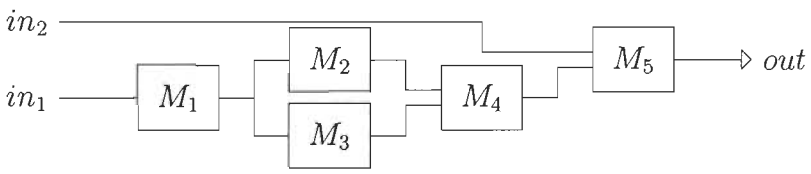
(17)



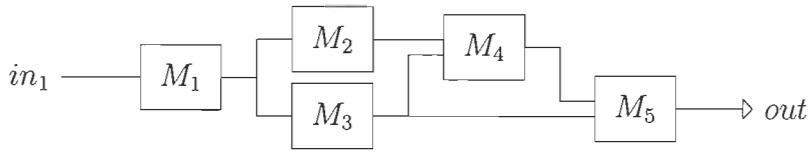
(18)



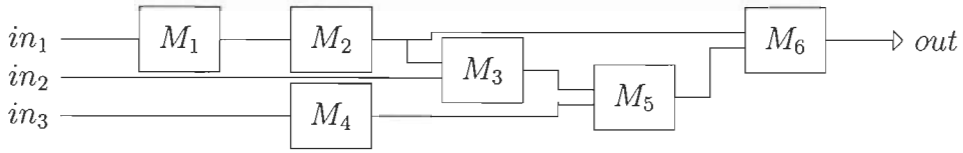
(19)



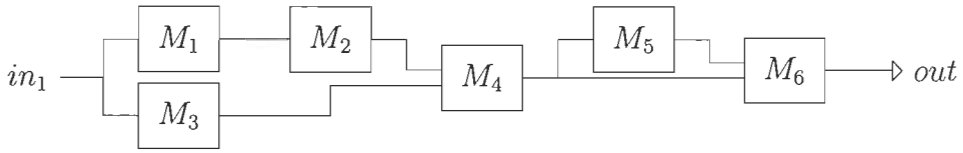
(20)



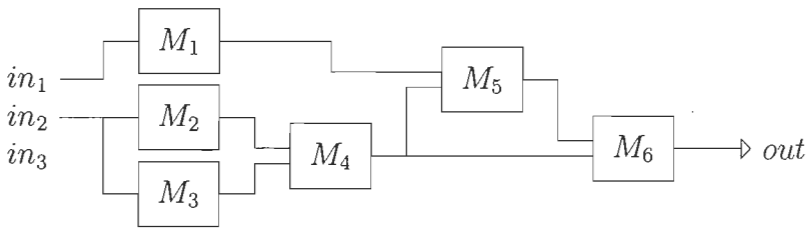
(21)



(22)



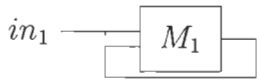
(23)



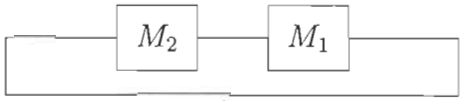
Annexe A.2 Chaînes de test non valides

Afin de déterminer le comportement du programme en cas de chaînes non valides, des chaînes de traitements, basées sur les 23 précédentes, ont été testées.

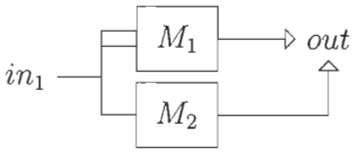
(1)



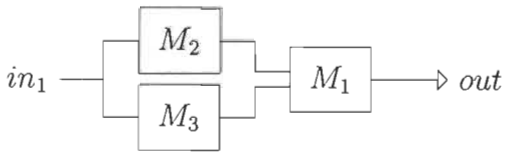
(2)



(3)

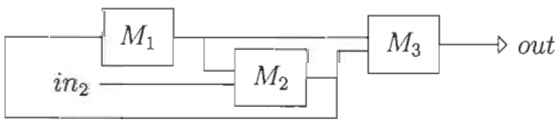


(4)

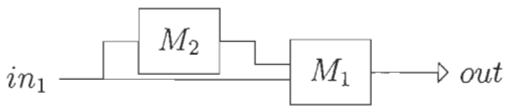


(incompatibilité de type entre la sortie de M_2 et l'entrée de M_1)

(5)

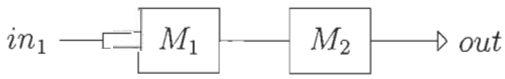


(6)



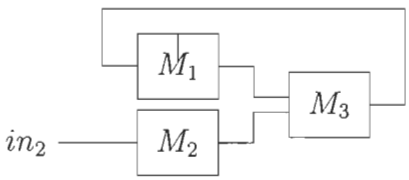
(incompatibilité de type entre l'entrée de M_2 et celle de M_1)

(7)

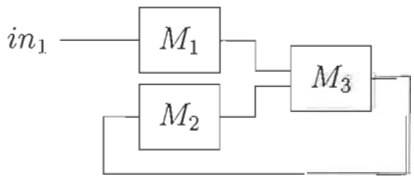


(incompatibilité de type entre les deux entrées de M_1)

(8)



(9)



Annexe A.3 Temps d'exécution

Les tableaux suivants présentent les temps d'éxécution exprimés en ms pour 500 passages pour chaque chaîne de test.

— Pour les chaînes de base

test	(1)	(2)	(3)	(4)	(5)	(6)
Dès que possible	11	19	18	31	62	107
Systématique	4	7	6	12	17	24

— Pour les chaînes courtes

test	(7)	(8)	(9)	(10)	(11)	(12)	(13)
Dès que possible	26	65	75	56	76	84	24
Systématique	27	21	22	27	32	18	8

— Pour les chaînes longues

test	(14)	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)
Dès que possible	152	97	125	245	197	140	159	307	140	184
Systématique	38	33	38	49	44	36	33	61	44	51