Rapport de stage de L3

Transformation automatique de phrases

Théo Zimmermann, directeur de stage : Paul Sabatier (DR CNRS)

LIF (Laboratoire d'Informatique Fondamentale – UMR Université d'Aix-Marseille/CNRS), équipe TALEP (Traitement automatique du langage écrit et parlé) - Marseille

Stage effectué du 4 juin au 3 août 2012

1 Introduction

Mon travail dans le cadre de ce stage avait pour objectif de rechercher un moyen, le plus général possible (notamment indépendant de toute grammaire), d'inférer des transformations de phrases à partir d'exemples (principalement des exemples de couples de phrases, phrase d'origine et transformée, mais aussi éventuellement des exemples négatifs), et d'en réaliser l'implémentation au sein du logiciel Illico. Par exemple, pour une phrase très simple comme « Max mange une pomme. », les transformations suivantes sont envisageables :

- La négation « Max ne mange pas une pomme. »
- Le passif « Une pomme est mangée par Max. »
- Différentes formes d'interrogations « Max mange-t-il une pomme ? » (totale), « Qui mange une pomme ? » (partielle)
- Pronominalisation « Il mange une pomme. », « Il la mange. », etc.

Cette problématique de transformation automatique de phrases s'inscrit dans le contexte plus général de l'analyse et de la génération de phrases sous contraintes dans le domaine du traitement automatique des langues. L'idée de transformation de phrases est à la base de théories linguistiques et syntaxiques comme celle de Zellig Harris (1952) qui se base sur les transformations autorisées dans une langue pour définir une relation d'équivalence sur les éléments de celle-ci (mots, phrases...). L'étude des transformations paraît être, en effet, une méthode appropriée pour comprendre une langue étant donné qu'elle sert aussi à apprendre une langue (maternelle ou étrangère) à des humains, à travers des exercices et jeux linguistiques.

Notons que nous entendons « transformation » dans un sens intuitif et général. Nous pourrions ainsi définir une transformation comme tout processus compréhensible et reproductible qui permet de passer d'une phrase à une autre. Harris (1995), en revanche, ne s'intéresse qu'à des transformations qui présentent un intérêt linguistique. Notamment, il pose deux conditions : que la transformation soit valable pour toute une classe de phrases et que le contenu sémantique ne change pas radicalement au cours de la transformation.

Le domaine du traitement automatique des langues est une branche de l'intelligence artificielle dont le but initial est la communication homme-machine (donner des ordres, interroger des bases de données en langue naturelle, la plus grande d'entre elles étant le Web) mais qui multiplie aussi les applications destinées à automatiser le traitement de textes, comme la traduction automatique ou la reformulation automatique. La transformation de phrases peut tout à fait servir dans cette dernière voie. Les fondateurs du domaine sont ceux de l'intelligence artificielle (parmi eux Alan Turing, 1950) ainsi que des linguistes (notamment Noam Chomsky, 1975, dont les travaux sur les langages formels ont des applications en informatique en dehors du domaine). Bien que j'ai pu être confronté parfois à des problèmes de type linguistique, mon travail était clairement centré sur les aspects informatiques de la question.

Le logiciel Illico, développé en Prolog au sein du LIF par Robert Pasero et Paul Sabatier (2007), permet d'analyser et de synthétiser des phrases dans une quelconque langue, une quelconque grammaire fournie par l'utilisateur (accompagnée d'un lexique et généralement d'informations sémantiques, conceptuelles, voire contextuelles). Le logiciel Illico permet en outre de définir des contraintes sur les phrases, et les transformations peuvent être vues comme des contraintes portant sur des couples de phrases, cependant ce

n'est pas l'approche qui prévaut dans le logiciel, les transformations étant toujours appliquées à une phrase pour en produire d'autres.

Illico fournit les différentes représentations formelles d'une phrase aux niveaux étudiés (lexical, syntaxique, sémantique, conceptuel et contextuel), sous forme de termes Prolog, c'est-à-dire d'arbres étiquetés. Les méthodes de transformation que nous implémentons se produisent au niveau de ces représentations, et non pas de la phrase elle-même comme nous aurions pu choisir de procéder. Cela nous permet bien évidemment d'être plus précis dans la définition de ces transformations et de limiter les classes de phrases sur lesquelles elles sont applicables. Notamment, les phrases à transformer et les phrases résultant de transformations font partie de l'ensemble des phrases bien formées que définit la grammaire de référence. C'est le logiciel qui, en fonctionnant en synthèse (ou « génération »), se charge ensuite de produire la ou les phrases qui constituent le résultat final de la transformation. Nous pouvons même profiter des capacités génératives d'Illico pour laisser certaines parties de la représentation libres (typiquement, des terminaux lexicaux pouvant prendre plusieurs formes ayant le même sens mais dont une seule est correcte, soit d'un point de vue grammatical, soit phonologique – choix entre « ne » et « n' » pour la négation, entre les différents pronoms personnels sujets pour la pronominalisation du sujet).

Le formalisme adopté au sein d'Illico pour la syntaxe est celui des grammaires de métamorphoses (Colmerauer, 1975) où les symboles non terminaux sont des termes (et pas seulement des atomes), formalisme lui même dérivé des grammaires de réécriture de Chomsky, et la représentation syntaxique sur laquelle je travaille est donc un arbre étiqueté dont les feuilles sont les terminaux de la grammaire et les éléments du lexique.

Les principaux tests auront lieu avec la grammaire GNF (grammaire noyau du français) développée par Pasero et Sabatier (2010).

* *
*

Dans la section 2 de ce rapport, je présente des remarques sur les choix qui ont été pris, par Pasero et Sabatier, puis par moi-même, et sur les alternatives qui auraient été envisageables. Dans la section 3, je résume le déroulé de mon stage de manière à peu près chronologique. Dans la section 4, je donne l'algorithme d'inférence auquel j'ai abouti. La section 5 me permet de présenter les résultats de mon travail (écriture de transformations et inférence). Je dresse des perspectives en section 6 et je conclus sur les résultats de mon travail de recherche en section 7. En section 8, je présente un autre aspect de mon stage : les discussions que j'ai eu avec les membres du laboratoire et les séminaires auxquels j'ai assisté. Enfin je donne des résultats détaillés et des informations supplémentaires sur mon travail dans les annexes et je termine avec une bibliographie.

2 Remarques et réflexions pour l'orientation du travail

Sur le choix du niveau de la représentation formelle

Une transformation de phrase n'est *a priori* pas restreinte au niveau syntaxique, mais c'est à ce niveau là que je me suis intéressé. Pour montrer l'intérêt qu'il y aurait à introduire des niveaux supplémentaires dans le processus, intéressons-nous à la pronominalisation du sujet. Cette opération transforme « Max dort. » en « il dort. ». « le chien par lequel Max est trouvé aboie. » admet deux transformées, qui sont « il aboie. » et « le chien par lequel il est trouvé aboie. ». En se restreignant au niveau syntaxique, la transformation « Max aime la fille à laquelle Bob pense. » en « Max aime la fille à laquelle il pense. » est valide. Il aurait fallu étudier le niveau sémantique pour se rendre compte que cette transformation n'est pas correcte (car elle rend ambigu le sens de la phrase).

De même, certaines transformations pourraient nécessiter des connaissances supplémentaires au niveau lexical. Igor Mel'čuk (1996) définit les fonctions lexicales dont l'objectif est justement de nous fournir ces connaissances idiomatiques. Par exemple, si nous souhaitons renforcer certains mots au sein d'une phrase, il faut être en mesure de sélectionner l'adjectif approprié. La fonction Magn le fournit. Mel'čuk et Wanner (2001) donnent les exemples suivants pour la langue anglaise :

- Magn(argument) = strong
- Magn(debate) = hot
- Magn(measures) = drastic
- Magn(heart attack) = severe

Le passif (par exemple « Max mange une pomme. » → « une pomme est mangée par Max. ») est un autre exemple de transformation qui ne peut être réduite au niveau syntaxique : nous avons besoin de connaissances supplémentaires pour déterminer le participe passé en fonction du verbe.

Sur le choix de la représentation d'une transformation

Nous écrivons systématiquement les transformations comme des relations entre les représentations formelles de différentes phrases. D'autres approches étaient envisageables. Au lieu de manipuler directement les représentations formelles, nous aurions pu expliciter une transformation comme un couple de contraintes liées entre elles et portant sur deux phrases, la phrase d'origine et sa transformée. Un exemple est l'utilisation de coupes syntaxiques. Nous pourrions profiter des capacités d'Illico à générer des phrases sous contrainte d'une coupe donnée (Pasero et Sabatier, 2012) pour obtenir le résultat de la transformation choisie. Les coupes sont des listes de terminaux et non-terminaux qui coupent l'arbre formant la représentation syntaxique de la phrase[1]. Pour pouvoir écrire les relations entre la phrase de départ et la phrase d'arrivée, il nous faut pouvoir retenir les sous-arbres correspondant à certains non-terminaux, par exemple en introduisant des variables. Cette fonctionnalité n'est pas offerte par Illico. Il faudrait donc l'y ajouter pour pouvoir tester la pertinence de telles représentations des transformations. Cependant, j'ai finalement observé qu'il semble plus puissant de représenter les transformations directement comme des contraintes sur des couples de phrases (ce que je fais) plutôt que des couples de contraintes sur des phrases, car on peut alors tirer parti des non-dits.

Pour illustrer ce propos, voici comment je représenterai la transformation d'une proposition dont le verbe n'a pas de complément en sa négation :

```
subtree(_,
  node(_,_,[
    node(neg_ne_ou_vide,2,[
      replace(<avec_ne>,tree(<sans_ne>)),
      replace(_,tree(<>))
    ]),
    node(neg_pas_ou_vide,2,
      replace(<avec_pas>,tree(<sans_pas>)),
      replace(<neg_pas,nil,pas>),tree(<>))
    ])
  ])
)
```

Telle qu'elle est écrite, elle permet de transformer à la fois une phrase comme « il vient. » mais aussi « vient-il? », « Max vient-il? » et même de rechercher la proposition à transformer en profondeur, comme par exemple c'est le cas pour « l'enfant qui vient mange une pomme. ».

Voici un duo de coupes augmentées permettant de transformer « il vient. » en « il ne vient pas. » :

```
Phrase de départ :
      [x,<>,y,z,<>,p]
Phrase d'arrivée :
      [x,neg_ne_ou_vide,y,z,<pas>,p]
```

L'écriture paraît plus simple. C'est surtout en raison de la syntaxe plus concise. Elle a cependant nécessité de prendre en compte des détails grammaticaux inintéressants pour la transformation mais nécessaires pour reproduire les parties inchangées comme le groupe nominal sujet `x`, le verbe `y`, le pronom postverbal `z` et le point de ponctuation `p`. De plus, la transformation décrite ici est moins générale : des différents exemples de phrases traitées dans le premier cas, seules « il vient. » et « Max vient-il? » le sont toujours.

1 La définition formelle d'une coupe est la suivante : Soit $A = r(a_1, ..., a_n)$ un arbre syntaxique avec $n >= 0$.
Une coupe syntaxique k de A est une séquence de symboles terminaux ou non-terminaux qui vérifie :
– soit $k=[r]$;
– ou $k=k_1¤ ...¤ k_n$, où chaque k_i est une coupe syntaxique de l'arbre a_i, et ¤ désigne la concaténation.

3 Déroulé du stage

Permettre à l'ordinateur de trouver par lui-même les transformations à effectuer à partir d'exemples, que ce soit par des mécanismes d'inférence logique ou d'apprentissage automatique statistique, nécessite au moins deux étapes. La première est de choisir une représentation claire et facilement manipulable par ordinateur de ce qu'est une transformation. Par conséquent, j'ai commencé par écrire des transformations à la main, qui m'ont permis d'aboutir à un formalisme. La deuxième est de construire l'algorithme réalisant l'inférence. J'ai utilisé au maximum les outils fournis par Prolog, algorithme essai-erreur, avec backtracking, contraintes, pour ensuite, devant certaines impasses ou difficultés, chercher à élargir mon point de vue par des recherches bibliographiques sur les différentes méthodes envisageables. Bien entendu, l'algorithme est directement lié à la représentation de la transformation, ce qui peut nous amener à la remettre en question.

Représentation des transformations

Jusqu'ici les transformations étaient écrites en fournissant pour chacune une représentation complète de la phrase d'origine et de la phrase d'arrivée (avec des parties libres représentées par des variables). Le problème principal était alors la variabilité de ces représentations pour une même transformation, qui donnait lieu à une multiplicité de règles. Par exemple, des verbes de différentes catégories (transitif direct, indirect, intransitif) nécessitaient parfois d'être traités séparément à cause de leurs représentations syntaxiques différentes.

Dans un premier temps, j'ai réalisé des fonctions de transformation d'arbre (trouver, remplacer, insérer, déplacer) qui m'ont permis de simplifier l'écriture de ces transformations de phrase. En particulier, elles permettent de rechercher le sous-arbre sur lequel appliquer la transformation en profondeur dans l'arbre syntaxique représentant une phrase. Cela a considérablement allégé l'écriture. Par ailleurs, ces fonctions permettent de traiter de nombreux cas particuliers en même temps et par conséquent réduisent le nombre de règles nécessaires à la définition d'une transformation donnée.

Devant la complexité de certaines transformations, j'étais sans cesse contraint d'écrire de nouvelles fonctions de manipulation d'arbre (par exemple, trouver un ensemble de fils communs d'un même nœud). J'ai alors réfléchi à une fonction qui serait suffisamment générale pour que toute transformation d'arbre se réduise à un appel à celle-ci. Ma fonction, `find_replace`, prend en paramètre un « motif de recherche » défini par une syntaxe spécifique, récursive, et la complexité repose désormais dans l'écriture de ce motif. Celui-ci peut comporter des mots-clés indiquant qu'il faut mémoriser un sous-arbre dans une variable, en remplacer un autre, reconnaître un nœud et rechercher en profondeur dans ses fils, mais aussi des opérateurs booléens. J'ai même introduit un opérateur `maybe` qui permet d'effectuer des transformations dans un sous-arbre uniquement lorsque celui-ci existe et qui m'a posé de nombreux problèmes par la suite notamment car il faut traiter séparément les cas d'un sous-arbre qui n'existe pas ou bien qui n'est pas de la bonne forme. Enfin, je me suis rendu compte qu'il me manquait un opérateur `forall` qui effectuerait des transformations dans tous les sous-arbres de la forme recherchée.

Grâce à cette fonction, la relation entre les représentations formelles d'une phrase et de sa transformée est décrite par une contrainte liant ces deux représentations entre elles (le motif de recherche). La syntaxe exacte telle qu'elle a été implémentée est fournie en annexe. Nous verrons cependant qu'elle n'est pas assez symétrique vis à vis des deux phrases et nous en proposerons une autre qui règle ce problème.

Cette fonction générale de transformation d'arbre ne ressemble pas à la manière standard dont de telles transformations sont habituellement envisagées. En effet, les problèmes de *tree edit distance*, inspirés par les problèmes de *string edit distance* (voir Akutsu, 2010), mettent en jeu les opérations fondamentales d'insertion, de suppression et de substitution (ré-étiquetage), et les composent pour obtenir toutes les transformations désirées. Quant à moi, je m'appuie sur l'opération fondamentale de remplacement et j'effectue toutes les transformations d'un seul coup durant l'exploration de l'arbre. Je laisse possible la composition, à l'aide de l'opérateur `and`. Toutefois, par la suite, je n'ai pas cherché à exploiter toute la richesse de la syntaxe que j'avais définie durant l'inférence d'une transformation. En particulier, l'opérateur `and` ne sera jamais utilisé automatiquement. Ce qui laisse ouverte la question (intéressante) de savoir s'il serait envisageable / facile / intéressant d'inférer des transformations en les représentant comme des composées d'opérations élémentaires de manipulation d'arbre. On notera que l'utilisation de `and` pour écrire

une transformation telle que le passage à la forme interrogative (voir en annexe) révèle une faiblesse de la représentation des transformations que j'ai implémentée et que celle-ci n'est pas totalement résolue dans la proposition que je fais finalement pour modifier cette représentation.

Inférence des transformations

Les bases sont alors posées pour passer à l'inférence automatique de nouvelles transformations. Je dispose d'une règle Prolog prenant en paramètres un motif de recherche détaillant la transformation qui doit avoir lieu, la représentation syntaxique d'une phrase de départ et capable de fournir la représentation syntaxique de la phrase d'arrivée. Il suffit que cette règle soit assez bien écrite pour pouvoir fonctionner en génération de motif de recherche (étant donné qu'en Prolog, on ne définit pas de fonction mais des relations, une relation suffisamment bien écrite peut aussi être utilisée pour retrouver des antécédents).

J'ai fait les quelques modifications que désirait mon directeur de stage sur l'interface du logiciel Illico (réalisée en Java, et la communication avec Prolog passant par une bibliothèque C), ce qui m'a permis de mieux appréhender le fonctionnement du logiciel.

J'ai alors préparé ce dernier pour cette nouvelle fonctionnalité : un menu permet de lancer l'algorithme d'inférence à partir de la donnée de fichiers comportant la liste des phrases de départ, la liste des phrases d'arrivée et la liste des phrases ne subissant pas la transformation recherchée. Plus tard, j'ai décidé de tout regrouper en un même fichier avec une syntaxe particulière, afin de faciliter l'utilisation.

Ces types d'exemples (transformations, phrases ne subissant pas la transformation) sont ceux suggérés par Pasero et Sabatier (2012). Je savais que dans un premier temps, je ne me servirais que des exemples de transformation. On notera que d'autres « exemples négatifs » à la définition d'une transformation peuvent être envisagés, comme le fait de fournir la liste exhaustive des transformées d'une certaine phrase (et donc d'exclure tout autre résultat). Il n'est cependant pas évident de trouver comment utiliser ces exemples négatifs. Cela l'est d'autant moins que j'ai toujours cherché à obtenir les motifs de recherche les plus spécifiques satisfaisant aux différents exemples positifs donnés ; je n'avais donc pas idée de ce que je pourrais préciser à l'aide de mes exemples négatifs. En fait, certains choix entre des motifs incomparables en terme de spécificité pourraient en dépendre.

Pour mes premières recherches, le résultat (motif trouvé) était seulement affiché. Pour le vérifier en le testant sur d'autres exemples notamment, il est plus pratique que la transformation obtenue soit immédiatement utilisable dans le logiciel. Cette idée est inspirée d'une fonctionnalité proche existant pour les contraintes portant sur une unique phrase : on peut lister les coupes sur un exemple de phrase, en choisir une, et générer d'autres phrases avec cette contrainte. Comme le motif de recherche constitue quasiment l'intégralité du code d'une transformation, il était très facile d'utiliser celui-ci pour en déduire la transformation. C'est ce que je ferai finalement de manière automatique au sein du logiciel, utilisant à l'occasion les propriétés de langage dynamique de Prolog pour rajouter des règles en cours d'exécution.

Recherches sur l'algorithme d'inférence

Bien sûr, à une transformation donnée peuvent correspondre un grand nombre de motifs de recherche corrects. Cela est d'autant plus vrai quand la transformation n'est définie qu'à partir d'exemples. J'ai choisi de ne jamais retenir que le premier motif trouvé et j'ai ordonné les règles définissant la relation `find_replace` de manière à ce que le motif trouvé soit le plus spécifique possible. J'ai alors obtenu des premiers résultats, logiques et inintéressants : pour une seule phrase d'entrée et une seule phrase de sortie, de représentation syntaxique respective `S` et `T`, le motif trouvé est naturellement `replace(T,tree(S))`, signifiant dans la syntaxe que j'ai définie « remplacer `S` par `T` ». En fait, cela ne restera pas vrai par la suite, car on essaie d'effectuer les remplacements uniquement lorsqu'ils sont nécessaires, c'est-à-dire le plus en profondeur possible.

J'ai installé des contraintes sur le motif pour accélérer sa génération mais elles m'ont beaucoup compliqué le travail et j'ai constaté dans les premiers temps que l'opération la plus longue était l'analyse des phrases d'exemple et non l'inférence à proprement parler. Je laisserai donc de côté cet usage des contraintes par la suite, pour enfin y revenir pour d'autres raisons.

Je ne chercherai pas non plus à générer des motifs contenant des opérateurs booléens, ce qui compliquerait exagérément la tâche pour un intérêt limité.

Je suis parvenu, après débogage sur des grammaires plus simples, dont une que j'ai créée pour l'occasion, à une première version de l'algorithme capable de trouver des motifs en recherchant en profondeur, parcourant des nœuds et la liste de leurs fils et effectuant des remplacements internes. Cette version s'est révélée satisfaisante pour des transformations simples telles que la suppression des « déterminatives » (propositions relatives déterminantes) ou encore certaines restrictions de la négation ou de la pronominalisation du sujet (voir les exemples en annexe).

Comme je souhaitais, dans cette première version de l'algorithme qui ne tient compte que des « contraintes positives » sur les transformations (des exemples de phrase et sa transformée), que les motifs soient systématiquement généralisés lorsqu'ils ne sont pas satisfaisants, pour tenter de recouvrir tous les exemples proposés, il me fallait notamment permettre de laisser libres certaines données comme la profondeur de recherche d'un sous-arbre, l'étiquette d'un nœud ou le nombre de ses fils. Pour cela, j'ai ajouté des points de choix dans les règles pour décider à quel point le motif doit être spécifique. Par la suite, je montrerai qu'on gagne beaucoup en complexité en procédant autrement, de manière à ne pas multiplier les points de choix et donc le backtracking durant la génération d'un motif. Pour éviter que de telles variables laissées libres soient ensuite liées par la vérification du motif sur les phrases qui suivent (la fonction `find_replace` utilisant fréquemment l'unification de termes Prolog, qui affecte y compris les « arguments » lors d'un appel à une règle), j'ai dupliqué le motif avant chaque test sur un autre exemple. J'en arrive donc à une étape où génération du motif et tests sur les exemples commencent à être clairement distingués.

Un problème se pose pour copier ou déplacer un sous-arbre. Dans la fonction de transformation d'arbre, j'utilise le couple de motifs `remember(x,f)` et `replace(x,f')`, équivalent d'un copier-coller. Je laisse alors à l'unification de Prolog le véritable travail : en effet, ma fonction se contente d'unifier la variable `x` au sous-arbre en cours d'exploration lorsqu'elle rencontre le motif `remember(x,f)` et au sous-arbre produit lorsqu'elle rencontre `replace(x,f')`. Non seulement cette méthode pose problème en génération car elle ne clarifie pas de quelle manière on devrait noter les éléments qu'il faut retenir, recopier, etc ; elle ne définit pas d'ordre dessus (copier puis coller) ; mais en plus, elle est responsable d'une utilisation supplémentaire des variables difficile à gérer lors de la création de motif, car à ce moment là, on ne peut pas utiliser la technique à base d'unification car elle modifierait le motif lui-même.

L'idée pour résoudre ce problème est tout d'abord d'ajouter des `remember` à chaque nœud et des `replace` là où ils sont nécessaires, mais en laissant libre ce par quoi on remplace ; pour ensuite, au cours d'une autre étape, tenter d'unifier entre elles les variables. On supprime finalement les `remember` inutiles (dont la variable n'a été unifiée à aucune autre). On peut voir que cette méthode nous encourage à considérer la plupart des parties de la phrase (même celles n'étant pas modifiées) dans le motif de recherche trouvé ; il faut les parcourir seulement pour ajouter des `remember`, ce qui nécessite de spécifier au plus le motif de recherche. Cela va nous conduire par ailleurs à considérer le motif `node(name,arity,sons)` comme plus spécifique que `subtree(depth,pattern)`, ce qui causera d'autres difficultés comme nous allons le voir par la suite.

Préalablement, j'ai voulu repenser ma fonction `find_replace` pour plus de symétrie entre l'arbre et son transformé. J'ai ajouté un pré-traitement permettant de passer d'un codage des arbres par des n-uplets à un codage à l'aide de listes, et un post-traitement pour revenir à un codage par n-uplets finalement. Dans les versions précédentes, la transformation d'un n-uplet en liste se faisait lors de la recherche, à chaque nœud rencontré. J'ai évité d'utiliser toutes les fonctions qui pouvaient générer des erreurs de type, par exemple lorsqu'on leur fournissait une variable libre. Notamment, j'ai créé ma propre relation pour donner la taille d'une liste qui peut aussi générer une liste d'après sa taille. L'idée de ces modifications est de permettre que l'on utilise la même fonction pour opérer une transformation et pour vérifier un motif lorsque la transformée est connue. Elles ont aussi rendu possible l'utilisation de la fonction `find_replace` « à l'envers » pour écrire facilement des transformations inverses (par exemple, le retour à l'affirmative).

Avec ma nouvelle fonction, je suis parvenu à obtenir toutes les fonctionnalités dont je disposais auparavant, mais avec une seule et même fonction pour la génération et la vérification, qui sert à la fois à l'inférence et aux transformations écrites à la main. Cependant, cette manière de procéder va de nouveau me

conduire à une impasse ou à séparer à nouveau génération et vérification dès que je me lancerai dans des objectifs plus compliqués (copier-coller, opérateurs booléens...). En effet, dans une approche où toutes les tâches sont ainsi intégrées, toute modification / amélioration de l'une menace de créer des erreurs dans les autres. Par ailleurs, elle est génératrice de complexité de l'algorithme qu'il est alors difficile d'optimiser.

Une autre approche finalement adoptée, utilisant toujours une unique fonction `find_replace`, consiste à réintroduire des contraintes, et utiliser une fonction indépendante de toute analyse de phrase pour générer le motif. Dans cette seconde approche, que j'ai commencé à tester, les contraintes préalables sur le motif sont indispensables. Notamment, si la fonction de génération de motif est trop simple (programmée de manière classique en Prolog, récursive non terminale), elle peut parcourir des branches infinies conduisant à des motifs de plus en plus longs avant même d'avoir énuméré tous les motifs de petite taille. Il est nécessaire par ailleurs de disposer d'une fonction spéciale pour dupliquer le motif au fur et à mesure de sa génération.

Cette seconde approche étant plus souple, les règles de génération pouvant être très différentes des règles de vérification, elle me permet d'ajouter facilement des `remember` aux emplacements voulus (c'est-à-dire presque partout).

A ce moment là, c'est la fonction spéciale de duplication qui se charge au passage de décider quelles variables devraient être laissées libres et d'unifier entre elles certaines variables afin d'établir les correspondances nécessaires au copier-coller par une méthode d'essai-erreur.

4 Version finale de l'algorithme

En utilisant cette dernière approche, il m'a suffi de rajouter de manière aveugle le plus possible de `remember` dans mon arbre de recherche pour obtenir un algorithme d'inférence qui réussisse aussi à déplacer des sous-arbres dans l'arbre sur lequel on effectue la recherche. Cependant, cet algorithme est trop long : une dizaine de minutes de calcul (sur une machine peu puissante) pour inférer une transformation décrite par les exemples suivants dans la grammaire GNF :

- Max aime Marie. → Marie aime Max.
- Max aime Eve. → Eve aime Max.

Si on fait varier le prénom masculin aussi, alors on n'obtient pas de résultat après plusieurs heures.

Le mécanisme principal permettant d'effectuer le « copier-coller » est la tentative d'unification des différentes variables libres entre elles. Celle-ci ayant lieu pendant la duplication du motif, elle est responsable d'une multiplication des points de choix, et donc des retours en arrière qui rallonge la découverte d'un premier motif. J'ai repoussé cette étape après la fin de la génération d'un motif de recherche en reprenant l'idée que j'avais eu au départ de tenir compte des retours des vérifications. Je ne cherche alors à unifier entre elles que les variables qui seraient restées unifiables après passage par la fonction `find_replace`. Sur l'exemple précédent, j'ai divisé par près de 4 la durée de calcul.

Lors de la duplication du motif chaque variable est remplacée par une autre afin d'éviter qu'une modification au cours de la vérification affecte le motif initial. Pour décider si les variables du motif doivent ou non rester libres, on tire parti de cette étape : la génération produit toujours des motifs dans lesquels les variables sont libres. Ensuite, lors de la duplication, selon qu'on remplace les variables ou non, celles-ci peuvent se retrouver liées au cours des vérifications à venir. Cette technique aussi est responsable d'un très grand nombre de points de choix et on a pu repousser la décision à plus tard ce qui a permis de réduire considérablement la complexité (toutes les transformations sont désormais inférées en moins de 10 millisecondes). En fait, comme la vérification a lieu au fur et à mesure de la génération du motif et que les étapes génératrices de points de choix inutiles ont été reportées, on est systématiquement en mesure de savoir si un choix de motif est le bon juste après l'avoir généré. C'est donc comme si nous avions des conditionnelles et que nous ne recourions jamais à du backtracking.

Désormais, l'algorithme est divisé en six étapes :

- établissement de contraintes sur un motif de recherche représenté par une variable libre à l'aide de la fonction `find_replace` non modifiée ;

- établissement de contraintes liant la variable précédente avec le motif qui sera généré, ces contraintes permettent de dupliquer le motif avant que ce dernier ne soit affecté par les vérifications ; cette étape renvoie par ailleurs une matrice détaillant pour chaque duplication par quelles variables les variables initiales ont été remplacées ;
- génération (sous les contraintes qui viennent d'être détaillées) d'un sous-ensemble de tous les motifs de recherche : ne sont pas considérés les opérateurs booléens ou associés, les `replace` lorsqu'ils deviennent inutiles ; les `remember` sont quant à eux ajoutés à chaque nœud où ils pourraient s'avérer utiles ;
- unification du plus grand nombre possible de variables aux variables correspondantes qui ont été liées durant la vérification : si une variable a été dupliquée n fois, cela suppose que les n variables correspondantes soient unifiables. Pour le vérifier, on cherche à unifier tous les éléments d'une même colonne de la matrice de correspondance ;
- unification du plus grand nombre possible de variables entre elles : si on unifie deux variables entre elles, cela suppose que pour chaque exemple cette unification aurait été possible. Pour le vérifier, on cherche à unifier élément par élément deux colonnes de la matrice de correspondance ;
- simplification du motif : on enlève les `remember` inutiles (dont la variable n'a été unifiée à aucune variable utile dans un `replace`) et les motifs qui sont trop généraux pour préciser quoi que ce soit.

Complexité

Les trois étapes initiales consistent à parcourir le motif en cours de création en effectuant à chaque fois les vérifications nécessaires pour faire un choix (sur lequel on ne reviendra pas). Ces vérifications consistent fréquemment à tester si les exemples d'arbres d'entrée et de sortie sont unifiables (opération linéaire en la taille du plus petit terme à unifier). La vérification la plus longue est due à `subtree` car on doit au préalable sélectionner une branche qu'on décidera d'explorer en profondeur, les autres devant être unifiées entre l'arbre d'entrée et celui de sortie. Ainsi on peut globalement (et grossièrement) borner la complexité du choix d'un mot-clé pour poursuivre la génération de motif par `O(d.T)` où `d` est le nombre maximal de fils qu'ont les nœuds dans les exemples fournis et `T` est la taille totale des représentations formelles des phrases d'exemple. Notons qu'on pourrait réduire cette complexité à `O(T)` à condition d'améliorer l'analyse de `subtree` par `find_replace`.

Par ailleurs, le nombre d'étapes de génération est égal à la taille finale du motif (ou plutôt la moitié de celle-ci si on compte les `remember` ajoutés à presque chaque étape) qui elle-même est bornée par `t`, soit la plus petite taille de la représentation formelle d'un exemple de phrase d'entrée. Finalement, on peut borner la complexité globale de ces trois étapes par `O(d.T.t)`.

La complexité des quatrième et cinquième étapes dépend du nombre de variables dans le motif trouvé. Or celles-ci ne sont pas plus nombreuses que `3t`. Elle dépend aussi de la taille des termes associés à ces variables, qui est toujours plus petite que celle des arbres de départ. Ces étapes sont globalement en `O(t.T)`.

La sixième et dernière étape est linéaire en la taille du motif trouvé. On a donc un algorithme polynomial. Cependant, ceci est rendu possible grâce à une limitation du backtracking due aux contraintes installées en amont. L'introduction de conditions devant être vérifiées une fois le motif généré et pouvant l'invalider, comme ce serait le cas si l'on tenait compte d'exemples négatifs de la manière la plus naïve qui soit, nous ferait immédiatement revenir à une complexité exponentielle.

Processus pour déterminer le motif

Les choix sont systématiquement faits de sorte à favoriser les motifs les plus spécifiques, pour généraliser ensuite selon la nécessité. Ainsi, moins un motif comporte de variables libres, plus il est précis. On cherche donc à leur donner une valeur ou à défaut à les unifier entre elles. Les motifs sont ordonnés comme suit, du plus spécifique au moins spécifique :

1. `tree(t)` donne le terme exact qui doit être trouvé,
2. `subtree(depth,pattern)` décrit à quelle profondeur dans les sous-arbres il faut aller rechercher le motif fourni,
3. `node(name,arity,sons)` détaille le nom du nœud, le nombre et le plus grand nombre possible de ses enfants,
4. `anything` ne précise rien quant à l'arbre en cours d'exploration mais indique que celui-ci n'est pas modifié durant la transformation,
5. `replace(x,pattern)` dit qu'il faut effectuer un remplacement, à condition d'avoir trouvé le motif fourni.

En fait, le choix d'ordonner `node` et `subtree` ainsi est arbitraire. En effet, ce sont deux mots-clés qui peuvent convenir pour décrire la même chose ou bien déboucher sur des motifs incomparables en terme de spécificité, en particulier lorsque la profondeur est laissée libre en ce qui concerne `subtree`. Tout l'intérêt de ce dernier mot-clé réside d'ailleurs dans cette possibilité. J'avais initialement choisi l'ordre inverse afin que le maximum de branches soient spécifiées, et notamment dans le but d'ajouter des `remember` dans ces branches. Cependant cela avait pour conséquence que `subtree` n'était jamais utilisé et que beaucoup moins de transformations étaient correctement inférées. Le choix actuel n'est pas idéal non plus. Je détaillerai dans les perspectives de nombreuses idées qui pourraient être utiles pour faciliter le choix et permettre que cet ordonnancement ne soit pas figé.

5 Résultats

Les améliorations successives que j'ai apporté à mes algorithmes avaient souvent pour finalité l'élargissement de l'ensemble des exemples sur lesquels on avait des résultats concluants.

Pour tester la puissance de ma fonction générale de manipulation d'arbre, j'ai écrit diverses transformations qui l'utilisent. Cependant, je me sers souvent de mots-clés tels que les opérateurs booléens (voire de l'opérateur de composition `and`) que je n'ai pas permis à mon mécanisme d'inférence d'utiliser. Par conséquent, ces transformations peuvent être plus générales que celles que l'on peut inférer.

J'ai écrit cinq transformations dont le code est donné en annexe :

- La suppression des propositions subordonnées relatives déterminantes dites « déterminatives » est celle dont le motif est le plus simple (une ligne) et n'utilise aucune fonctionnalité avancée. Par conséquent, le logiciel parvient à l'inférer parfaitement par la suite.
- La négation regroupe deux cas en un seul motif à l'aide de l'opérateur `maybe` qui sert à traiter le cas où le verbe a un complément (dans GNF, des traits de ce complément indiquent si le verbe dont il dépend est à la forme négative). Le logiciel a été capable d'inférer ces deux cas de la transformation séparément. Par contre, grâce à la possibilité de remplacer un sous-arbre par une variable libre et d'utiliser ensuite les capacités génératives d'Illico, il n'est pas nécessaire de distinguer les cas où la négation s'écrit avec « ne » et ceux où elle s'écrit avec « n' ».
- Le retour à l'affirmative a été l'occasion pour moi de tester la relative symétrie de la fonction `find_replace`.
- La pronominalisation du sujet profite elle aussi de pouvoir laisser des variables libres pour laisser à Illico le choix entre le pronom « il » et « elle ». En revanche, il a fallu distinguer ces cas pour l'inférence afin d'obtenir des résultats. Par ailleurs, j'ai écrit une transformation qui est assez générale pour traiter le cas par exemple d'une phrase telle que « chaque personne applaudit. ». Mais pour cela j'ai utilisé les opérateurs `and` et `maybe`.
- Le passage à la forme interrogative m'a demandé de ruser en utilisant très largement la composition avec `and` ainsi qu'une astuce impossible à inférer consistant à remplacer par un terme Prolog incomplet, c'est-à-dire contenant des variables libres elles-même destinées à être unifiées à d'autres termes plus tard, par exemple à l'aide de `remember`. Cette transformation fait partie de celles qui

ont été impossible à inférer et me conduit à proposer une nouvelle version de `find_replace`, plus symétrique, qui aurait aussi des conséquences sur l'algorithme d'inférence.

Pour tester l'étendue de mon algorithme d'inférence, j'ai cherché à construire automatiquement diverses transformations, à chaque fois avec les exemples les moins nombreux et les plus généraux possibles. Tous les résultats détaillés sont donnés en annexe. Outre ceux déjà cités, j'ai obtenu des résultats positifs pour l'inversion du sujet et du complément lorsque ce sont deux noms propres. Cette transformation m'a permis de tester la réussite de l'inférence du « copier-coller ».

En revanche, je n'ai pas réussi à inférer l'inversion du sujet et du verbe dans une proposition relative suivant l'exemple :

« l'enfant que Max photographie dort. » → « l'enfant que photographie Max dort. »

Le problème soulevé par une telle transformation est du même type que celui posé par l'interrogative.

Concernant la passivation, je m'attendais à un résultat négatif puisque cette transformation n'est pas purement syntaxique.

6 Perspectives

J'ai expliqué précédemment le problème du choix dans l'algorithme d'inférence entre les mots-clés `subtree` et `node`. En effet, on ne peut pas décider si l'un est plus spécifique que l'autre mais il serait bon de disposer d'un moyen de faire un choix au plus tôt entre les deux. C'est particulièrement important car ce choix sera souvent définitif. En fait, il le sera tant qu'on n'aura pas introduit un moyen de tenir compte des exemples négatifs qui seuls peuvent permettre d'exclure certains motifs qui semblent convenir.

Voici des pistes pour la décision :

- `node` permet en apparence d'être plus précis. On pourrait vérifier que les précisions apportées dans les branches inchangées sont effectives, ou bien que ces branches doivent être mémorisées à l'aide d'un `remember` ;
- on pourrait obliger les profondeurs de recherche pour `subtree` à ne pas être toutes égales ;
- on pourrait fournir des informations supplémentaires destinées à aider le choix de l'algorithme (par exemple, préciser le type de nœud au niveau duquel a lieu la transformation, ou encore quels sont les terminaux qui doivent impérativement être précisés dans le motif de recherche...) ;
- on pourrait faire un choix probabiliste tenant compte de plusieurs de ces critères.

Parallèlement, il serait utile de tenir compte enfin d'exemples négatifs qui peuvent être de plusieurs formes, tel que je l'ai expliqué précédemment. Bien entendu la méthode la plus simple consiste à tester ces exemples une fois un motif entièrement trouvé. Cette méthode est problématique au niveau de la complexité car elle nous fait de nouveau utiliser un algorithme qui liste les possibilités avant de les tester (évidemment de complexité exponentielle). Il faudrait introduire de nouvelles contraintes qui permettraient de tester au plus tôt la compatibilité avec ces exemples négatifs. Enfin, une question qui a son importance est de déterminer quels types d'exemples négatifs seraient pertinents pour améliorer l'inférence d'une transformation.

Comme je l'ai dit, certaines transformations sont actuellement impossible à inférer. C'est le cas de l'interrogative et de toute transformation qui nécessite de modifier la structure globale de la phrase en réutilisant les composants élémentaires. Le problème vient du fait que la modification de l'arbre syntaxique représentant la phrase est basée sur le remplacement d'un sous-arbre par un terme complètement explicité ou une variable qui peut être liée par ailleurs à l'occasion d'un `remember`. Pour être en mesure de procéder à des remplacements plus complexes, il faudrait que ce par quoi on remplace soit lui-même un motif qui nous permette de laisser certaines parties libres et de préciser certaines autres. Je donne en annexe un exemple d'une syntaxe qui suit cette idée. Ce changement de vue a une autre « bonne » conséquence : il rend entièrement symétrique la fonction `find_replace`. Cependant, un tel changement n'est pas suffisant pour réussir à décrire suffisamment simplement l'interrogative. En effet, utiliser `diverge` (équivalent dans cette syntaxe de `replace`) précocement revient à décrire la transformation comme un

couple de contraintes et comme nous l'avons déjà dit, cela ne nous permet plus de tirer parti des non-dits pour décrire une transformation suffisamment générale.

Les transformations d'arbres telles que nous les représentons se basent, nous l'avons dit, sur l'opération de remplacement (ou bien sur le concept de divergence dans l'amélioration proposée, mais cela revient au même : deux nœuds commencent à diverger dès lors qu'ils n'ont pas la même arité ou la même étiquette). On pourrait aussi explorer d'autres pistes, par exemple rajouter les opérations d'insertion et de suppression de branches. Ces dernières sembleraient particulièrement adaptées dans le cas de grammaires ayant une forte variabilité de l'arité des nœuds. On pourrait aussi chercher à décomposer une transformation en transformations élémentaires. Cette piste pourrait peut-être permettre de régler le problème de l'interrogative. Cependant, il faudra trouver un équilibre entre améliorer la puissance de description des motifs de recherche et ne pas rendre trop complexe l'algorithme d'inférence.

Il serait très utile de regarder d'autres niveaux : notamment sémantique et en particulier pour faire de la paraphrase (voir l'intérêt pour la paraphrase soulevé dans l'introduction de Mel'čuk, 1988). On pourrait commencer par imposer pour certaines transformations que la représentation sémantique soit invariante ou au moins que le contenu sémantique soit inchangé. On pourrait alors essayer d'écrire / inférer la transformation au passif en profitant de cette contrainte supplémentaire, des capacités génératives d'Illico et du faible nombre de synonymes présents dans GNF afin qu'Illico trouve tout seul le bon participe.

Ces algorithmes ont été développés en Prolog pour Illico mais ne sont dépendants ni de l'un ni de l'autre. Ils utilisent des mécanismes tels que l'unification qui sont facilement reproductibles en C et n'abusent pas du backtraking. Ils profitent parfois de l'unité au sein d'Illico du module d'analyse / synthèse pour compléter judicieusement des trous dans la transformée, unité qui d'ailleurs fait de ce logiciel un outil particulièrement adapté à l'étude des transformations. Tout logiciel qui en reprendrait le principe serait tout aussi adapté. Mais l'algorithme serait encore utilisable (dans la plupart des cas) sans cette unité. En revanche, ces algorithmes dépendent encore trop fortement du type de grammaires étudiées : le choix de consacrer le remplacement comme base plutôt que l'insertion / suppression de branches en est responsable. Il serait donc intéressant de voir comment les adapter au cas des grammaires de dépendance par exemple (Hays, 1960). Les grammaires de dépendance sont organisées différemment : les lexèmes ne se trouvent pas aux feuilles de l'arbre syntaxique mais comme étiquettes de tous les nœuds.

7 Conclusion

J'ai développé durant mon stage, en suivant les conseils de Paul Sabatier, des algorithmes qui m'ont permis d'obtenir d'intéressants résultats sur l'inférence des transformations. J'ai cherché à présenter ce travail de manière la plus générale et théorique possible. L'idée générale était de transformer des phrases en manipulant leur représentation formelle en tant qu'arbre. Ma démarche s'est divisé en deux parties : créer des outils de manipulation d'arbres que j'ai pu ensuite utiliser pour écrire le code de différentes transformations de phrases à la main, puis un algorithme pour inférer ces manipulations de manière automatique à partir d'une liste d'exemples. Cet algorithme parvient à inférer des types de transformations très variés, y compris des transformations impliquant de déplacer des sous-arbres à travers la représentation formelle de la phrase.

La durée de mon stage était trop limitée pour que je puisse aller aussi loin que je l'aurais souhaité. J'ai atteint néanmoins mon objectif initial puisque je dispose d'un algorithme d'inférence et de résultats significatifs. Avec quelques semaines de plus, j'aurais cherché à implémenter les idées d'amélioration de la représentation des transformations que je propose dans la section précédente. Dans une optique de plus long terme, je pense qu'il est important de s'attaquer maintenant à rajouter la dimension sémantique dans les transformations. Une des applications en vue serait alors de faire de la reformulation de textes à partir d'exemples (de textes et de leurs équivalents reformulés, dont on extrairait les transformations qui ont été appliquées).

En accord avec mon directeur de stage, je reviendrai en automne au LIF pour présenter à mon tour ce travail dans un séminaire interne de l'équipe.

8 Vie du laboratoire

Discussions avec les membres du laboratoire

Étant intéressé par l'intelligence artificielle, thème qu'on n'aborde pas ou peu à l'ENS, j'ai profité de mon arrivée dans ce laboratoire (issu historiquement du GIA – groupe d'intelligence artificielle – dans lequel Alain Colmerauer et son équipe ont notamment inventé Prolog), pour avoir des discussions sur le sujet avec mon directeur de stage mais aussi avec son collège Robert Pasero aujourd'hui à la retraite et avec un ancien étudiant à lui, Célestin Sedogbo. Je leur ai demandé de m'expliquer précisément ce qu'était l'intelligence artificielle, domaine vaste de l'informatique, mais aussi de m'indiquer ce qui s'y faisait actuellement.

Par exemple Célestin Sedogbo m'a dit que le système Siri de l'IPhone qui reconnaît et exécute les ordres qu'on lui donne à l'oral a été développé à Standford à la fois avec des outils d'apprentissage statistique mais aussi, paraîtrait-il, en utilisant le langage Prolog.

Paul Sabatier m'a prêté des livres pour m'introduire au sujet (*The Handbook of Artificial Intelligence*, *Formalizing Common Sense – papers by John Mc Carthy...*) dont j'ai fait usage, surtout en dehors des heures de mon stage car j'étais déjà assez occupé alors par ma recherche.

Outre ces discussions qui sortaient du sujet de mon stage, j'ai aussi rencontré des membres du laboratoire qui m'ont aidé à y voir plus clair, et surtout à avoir une vision plus large de mon travail.

En premier lieu, j'ai rencontré Alexis Nasr, directeur de l'équipe TALEP, dont les outils de prédilection sont très différents de ceux de Paul Sabatier, puisqu'il m'a notamment fait découvrir les grammaires de dépendance et la théorie sens-texte de Mel'čuk.

Quand j'ai voulu mesurer la complexité de mes algorithmes, j'étais démuni car tels qu'ils étaient écrits en Prolog, cela me paraissait une tâche difficile, et en tout cas pour moi inhabituelle. Notamment l'usage du backtraking rendait compliquée cette mesure, à moins de dire simplement qu'on pouvait borner la complexité par une exponentielle. Yann Vaxès, de l'équipe ACRO, m'a beaucoup aidé à ce moment-là. Je lui ai expliqué mon travail et il m'a aidé à prendre du recul. C'est à ce moment-là que j'ai réalisé que j'étais arrivé à un stade où l'usage du backtracking était devenu très limité et donc que j'avais un algorithme de complexité polynomiale, mais aussi que j'allais avoir des problèmes pour ordonner `subtree` et `node` selon leur spécificité.

Je n'ai encore jamais précisé dans ce rapport que ce stage avait aussi été l'occasion pour moi d'approfondir ma connaissance de Prolog (qui jusqu'ici se limitait aux bases). Parfois, c'est aussi en discutant avec des chercheurs, parmi lesquels Pascal Préa, que j'ai pu ouvrir les yeux sur certains aspects de ce langage très différent de tout ce que j'avais manipulé avant.

Séminaires

J'ai aussi profité de mon stage pour assister aux séminaires qui étaient organisés par les différentes équipes du laboratoire, me rendant pour cela par trois fois à Château-Gombert, à l'autre bout de la ville, pôle sur lequel est présent une partie de l'unité de recherche. Au-delà de l'intérêt que présentait pour moi le contenu de ces conférences, j'y ai mieux compris le fonctionnement de la recherche académique.

Séminaires de TALEP

C'est l'équipe dans laquelle je fais mon stage.

- *Reconnaissance automatique de relations causales* par Cécile Grivaz (Ecole Polytechnique Fédérale de Lausanne, Suisse) : cette présentation de son travail en thèse était originale car il s'agissait de présenter des résultats négatifs, voire des absences de résultats concluants avec les techniques « état de l'art » actuelles. Des raisons pouvant expliquer cette absence de résultats étaient la nécessaire connaissance du monde ou la haute subjectivité qui entrent en compte dans la détection de relations causales.

– *Traitement automatique du Tunisien* en passant par l'arabe standard par Ahmed Hamdi et Rahma Boujelbane, étudiants en thèse dans l'équipe : ils utilisent des transformations mais uniquement au niveau lexical.

Ces deux conférences m'ont permis de mieux comprendre en quoi consistait la recherche dans le domaine du traitement automatique des langues, domaine fortement connecté à la linguistique, bien que pour ma part, j'ai été assez peu préoccupé par des questions linguistiques durant mon stage de recherche en informatique.

Séminaires d'ACRO

Équipe algorithmique, combinatoire et recherche opérationnelle.

– *Finding an odd hole through two vertices of a planar graph in polynomial time*, Marcin Kamiński (Université Libre de Bruxelles) : outre le grand nombre de notions qu'il me manquait, les applications d'un tel travail étaient difficiles à cerner.

– *Chemins auto-évitants fractals*, Pascal Préa : le concept était déjà plus accessible (auto-évitant car un chemin ne doit pas se rencontrer lui-même ; fractal au sens où on passe d'une échelle à une plus petite en remplaçant un segment du chemin par un nouveau chemin auto-évitant de plus petite taille).

– puis Yann Vaxès sur le chapitre 7, *Flot sous-modulaire et applications*, du livre de Lau, Ravi et Singh : manifestement, les chercheurs étudient ensemble des notions avancées qu'ils ne connaissent pas bien. Ce compte-rendu de lecture était absolument inabordable pour moi qui n'avait pas été présent à ceux des précédents chapitres.

– Thèse de Daniela Maftuleac, *Algorithmes pour des complexes planaires de courbure non-positive*, directeur Victor CHEPOI : l'exposé était intéressant. La thésarde a ensuite séché sur quasiment toutes les questions qui lui étaient posées, mais qui sortaient toutes assez du sujet. Cela m'a convaincu de l'importance dans tout exposé de recherche de se préparer à ce genre de questions en maîtrisant bien les à-côtés de son sujet.

Séminaires de QARMA

Équipe apprentissage et multimédia.

– *Classification non supervisée dans les modèles mixtes fonctionnels*, Madison Giacofci, Laboratoire Jean Kuntzmann : techniques basées sur des ondelettes, très difficiles d’accès.

– *Vers l'apprentissage d'une intelligence socio-émotionnelle pour les personnages virtuels*, Magalie Ochs, LTCI, Telecom ParisTech : le cas du sourire. L'exposé était accessible et intéressant, notamment pour les ponts jetés avec la psychologie. Cette étude pratique avait consisté à créer des personnages souriants avec l'aide d'utilisateurs pour ensuite obtenir des résultats significatifs dans l'amélioration de la perception de la machine par l'utilisateur.

Séminaire de MOVE

Équipe modélisation et vérification.

– *Emptiness and Universality Problems in Timed Automata with Positive Frequency*, Amélie Stainer (INRIA Rennes) : grâce à mes recherches antérieures sur les automates temporisés, pour le cours de Langage formels suivi cette année, et mon intérêt pour le sujet, j'ai compris en bonne partie le travail de thèse présenté ici, même si l'exposé n'était pas toujours limpide. Par exemple, on conserve la condition d'acceptation de Büchi mais en y ajoutant une contrainte sur la fréquence de passage dans les états acceptants qui doit être strictement positive. On utilise le même type d'abstraction (automate des régions) légèrement modifié.

Annexes

Syntaxe de `find_replace`

`find_replace` prend en arguments un motif de recherche dont la syntaxe est spécifique et deux arbres Prolog quelconques `t` et `t'` :

- `anything` signifie que le motif recherché est quelconque mais que l'arbre et son transformé sont les mêmes : on unifie `t` et `t'`.
- `remember(x,f)` signifie qu'on recherche le motif `f` mais qu'on veut retenir l'arbre en cours d'exploration dans `x` : on unifie `x` et `t` et on poursuit la recherche avec `f`.
- `replace(x,f)` signifie qu'on recherche le motif `f` dans l'arbre de départ `t` et que l'arbre d'arrivée `t'` correspond à `x` avec lequel on l'unifie.
- `maybe(f)` signifie qu'on n'est pas sûr de trouver le motif `f` recherché. Si on ne le trouve pas, on unifie `t` et `t'` comme pour `anything`. Dans le cas où `maybe(f)` est utilisé pour décrire un fils d'un `node`, on teste le cas où le fils existerait effectivement et celui où il n'existe pas du tout.
- `no(f)` vérifie qu'on ne peut pas trouver le motif `f` puis unifie `t` et `t'` comme pour `anything`.
- `or(f1,f2)` essaie de rechercher les deux motifs `f1` puis `f2` en créant un point de choix.
- `and(f1,f2)` est un opérateur de composition : il recherche le motif `f1` dans `t` et le motif `f2` dans l'arbre résultant de la recherche et de la transformation de `t` par `f1`, afin d'obtenir `t'` finalement.
- `node(r,a,s)` signifie que `t` et `t'` sont des nœuds dont la racine est `r` (donc nécessairement commune à `t` et `t'` même si on choisit de laisser `r` libre), le nombre de fils est `a` (idem) et une sous-liste des fils est décrite par les motifs de recherche de `s`. On essaie de toutes les façons possibles de rechercher les motifs de `s` parmi les branches de `t` et `t'`. Les branches non concernées étant unifiées entre elles. Si un motif de `s` est de la forme `maybe(f)` on teste les deux cas où le motif `f` ferait partie de la liste des fils, ou bien où il n'en ferait pas partie.
- `subtree(d,f)` signifie que `f` doit être recherché dans un sous-arbre de `t` et `t'` dont la racine est à la profondeur `d` (la racine de `t` et `t'` étant à la profondeur 0). Tout l'intérêt d'un tel motif apparaît lorsque `d` est laissé libre.
- `tree(x)` signifie que les arbres `t` et `t'` sont les mêmes et sont égaux à `x` : unifie `x`, `t` et `t'`.

Les arbres Prolog sont représentés de manière standard à l'aide de n-uplets. Comme la manipulation de tels objets est délicate quand n varie, nous convertissons préalablement les arbres sur lesquels a lieu la recherche en remplaçant tout n-uplet par la liste correspondante. Pour faciliter l'écriture de motif à la main, les motifs `remember(x,f)`, `replace(x,f)` et `tree(x)` autorisent à fournir `x` soit dans sa représentation à l'aide de listes, soit dans la représentation standard à l'aide de n-uplets. Dans ce dernier cas, la conversion est effectuée avant de procéder aux unifications, en toute transparence pour l'utilisateur.

Par conséquent, les motifs écrits à la main utiliseront la représentation standard en n-uplets tandis que ceux générés automatiquement utilisent la représentation en liste plus pratique à manipuler.

Code des transformations écrites à la main

Suppression des déterminatives :

```
transfo:regle_trf(1,a,a') ->
  tree:find_replace( subtree(_,
    node(conjonction_de_determinatives_ou_vide,2,[
      node(_,1,[replace(vide,tree(non_vide))]),
      replace(<>,anything)
```

```
    ])
  ), a,a') ;
```

Négation :

```
transfo:regle_trf(2,a,a') ->
  tree:find_replace( subtree(_,node(_,_,[
    node(neg_ne_ou_vide,2,[
      replace(<avec_ne>,tree(<sans_ne>)),
      replace(_,tree(<>))
    ]),
    node(neg_pas_ou_vide,2,[
      replace(<avec_pas>,tree(<sans_pas>)),
      replace(neg_pas(<>,<pas>),tree(<>))
    ]),
    maybe(node(groupe_nominal,_,[node(_,_,[replace(avec_ne,tree(sans_ne))])]))
  ])), a,a' ) ;
```

Retour à l'affirmative :

```
transfo:regle_trf(3,a,a') ->
  transfo:regle_trf(2,a',a) ;
```

Pronominalisation du sujet :

On présente seulement le motif de recherche pour la lisibilité :

```
subtree(_,node(_,_,[
  replace(
    :groupe_nominal(
      <sujet.t1,3,_,pronom_personnel,_,_,_>,
      :pronom_personnel_sujet(<sujet.t1,3,preverbal>,_)
    ),
    node(groupe_nominal,_,[
      node(sujet.t1,6,[
        anything,anything,
        and(no(tree(pronom_indefini)),no(tree(pronom_personnel))),
        anything,anything,anything
      ])
    ])
  ),
  and(
    maybe(subtree(1,node(_,_,[replace(sans_chaque,tree(avec_chaque))]))),
    maybe(node(groupe_verbal,_,[
      node(_,_,[replace(sans_neg,tree(avec_neg))]),
      node(neg_pas_ou_vide,2,[
        replace(<avec_pas>,tree(<sans_pas>)),
        replace(neg_pas(<>,<pas>),tree(<>))
      ])
    ]))
  )
]))
```

Interrogative :

Voici la transformation qu'il est dans l'état actuel, impossible de reproduire par inférence :

```
subtree(_,and(
  replace(
    :interrogative(<>,:interrogative_totale(<>,x,_)),
    node(declarative,_,[remember(x,node(proposition,_,[]))])),
  subtree(2,and(
    or(
      replace(
        :proposition(<t1,:avec_pronom_sujet_postverbal>,g_v),
        node(proposition,_,[
          node(t1,_,[tree(sans_pronom_sujet_postverbal)]),
          node(groupe_nominal,_,[
            node(_,_,[
              anything,anything,
              tree(pronom_personnel),
              anything,anything,anything
```

```
            ])
          ]),
          remember(g_v,anything)
        ])
      ),
      replace(
        :proposition(<t1,:avec_pronom_sujet_postverbal>,g_n,g_v),
        node(proposition,_,[
          node(t1,_,[tree(sans_pronom_sujet_postverbal)]),
          remember(g_n,node(groupe_nominal,_,[
            node(_,_,[
              anything,anything,
              no(tree(pronom_personnel)),
              anything,anything,anything
            ])
          ])),
          remember(g_v,anything)
        ])
      )
    ),
    subtree(1,node(groupe_verbal,_,[
      node(_,_,[
        replace(avec_pronom_sujet_postverbal,tree(sans_pronom_sujet_postverbal))
      ]),
      node(_,_,[
        node(_,_,[
          replace(
            avec_pronom_sujet_postverbal,
            tree(sans_pronom_sujet_postverbal)
          )
        ])
      ]),   % verbe
      replace(_,node(pronom_sujet_postverbal_ou_vide,_,[]))
    ]))
  ))
))
```

Syntaxe proposée comme amélioration et exemple d'application

Le motif est censé décrire une contrainte qui porte sur deux arbres à la fois. Il tire toujours partie des non-dits pour signifier que les deux sous-arbres sont quelconques mais égaux. La différence tient à ce que, dès lors que les deux arbres divergent au niveau de leur racine, on décrit chacun des deux séparément, mais à l'aide d'une syntaxe commune. Inversement, lorsqu'ils sont entièrement égaux, on peut éventuellement les décrire (afin d'être spécifique) ensemble, avec cette même syntaxe.

Motifs pour décrire les contraintes sur un couple d'arbres :

- `equal(f)` est utilisé dès lors que les deux arbres sont égaux afin d'être spécifique. `f` est un motif décrivant un seul arbre.
- `node(r,a,s)` a le même sens qu'avant mais son utilisation ici implique qu'une des branches au moins présentera une divergence dans le couple.
- `subtree(d,f)` signifie qu'un seul sous-arbre diverge et celui-ci est à une profondeur `d`.
- `diverge(f1,f2)` est utilisé dès que la racine des deux arbres est différente (étiquette ou nombre de fils). `f1` et `f2` décrivent chacun un seul arbre du couple.

Motifs servant à décrire les contraintes portant sur un seul arbre :

- `tree(t)` pour préciser ce qu'est l'arbre exactement.
- `leaf` peut éventuellement servir à préciser qu'on a affaire à une feuille quelconque.
- `node(r,a,s)` a toujours le même sens. Les fils non spécifiés sont laissés libres.
- `anything` aussi.
- `remember(x,f)` signifie qu'il faut rechercher le motif `f` puis unifier l'arbre à `x`. Ce mot-clé sert

alors à la fois à retenir certaines parties de l'arbre mais aussi à effectuer les remplacements.

Exemple de motif écrit à la main avec la syntaxe améliorée pour l'inversion sujet-verbe dans une proposition relative

```
subtree(_,diverge(
  node(proposition_moins_relative,3,[
    remember(x,anything),
    remember(y,node(groupe_nominal,_,[]),
    node(groupe_verbal_moins,5,[
      anything,
      remember(a,node(neg_ne_ou_vide,_,[])),
      remember(b,anything),                          % verbe
      anything,
      remember(c,node(neg_pas_ou_vide,_,[]))
    ])
  ]),
  node(proposition_moins_relative,2,[
    remember(x,anything),
    node(groupe_verbal_moins_avec_inversion_du_sujet,5,[
      anything,
      remember(a,node(neg_ne_ou_vide,_,[])),
      remember(b,anything),                          % verbe
      remember(c,node(neg_pas_ou_vide,_,[])),
      remember(y,node(groupe_nominal,_,[])
    ])
  ])
))
```

Tests de l'algorithme d'inférence

Exemples pour l'inférence, exemples de tests, motif trouvé.

Suppression des déterminatives

Exemples fournis pour l'inférence

« ce chien regarde le chat qui mange la pomme qui appartient à Léa. »

→ « ce chien regarde le chat qui mange la pomme. »

« la fille au frère de qui nous parlons applaudit. » → « la fille applaudit. »

Tests concluants

« le chat auquel je donne une pomme regarde-t-il Léa? » → « le chat regarde-t-il Léa? »

« la fille par laquelle il est mangé applaudit. » → « la fille applaudit. »

Note

Souvent, on pourra constater que les transformations inférées marchent également avec des phrases à la forme interrogative alors qu'il n'y en avait pas dans les exemples fournis. En revanche, nous faisons en sorte que les transformations n'aient pas toujours lieu dans la proposition principale, et c'est cela qui rend notre motif trouvé suffisamment général.

Motif inféré

```
subtree(v37279,node(conjonction_de_determinatives_ou_vide,2,subtree(1,replace(vide,tree(
non_vide))).replace(nil,node(conjonction_de_determinatives,2,node(v65540,0,nil).node(det
erminative,2,node(v65540,0,nil).node(proposition_relative,3,node(v65540,0,nil).node(grou
pe_relatif,v59906,node(v65540,2,anything.tree(dif_de_de_qui_ou_duquel).nil).nil).node(pr
oposition_moins_relative,v50879,nil).nil).nil).nil)).nil))
```

Négation d'un verbe sans complément

Exemples fournis pour l'inférence

« Eve dort. » → « Eve ne dort pas. »

« la fille qu'il aime applaudit. » → « la fille qu'il n'aime pas applaudit. »

Tests concluants

« le chat qui miaule dort-il? » se transforme en « le chat qui ne miaule pas dort-il? » et « le chat qui miaule ne dort-il pas? ».

« la pomme qu'il mange est verte. » → « la pomme qu'il ne mange pas est verte. »

Note

On remarque que ce n'est pas la qualité transitive ou intransitive du verbe qui change la manière dont il faut le traiter mais bien s'il est suivi ou non d'un complément. Ceci est dû à la manière dont est faite la grammaire GNF (le groupe nominal complément du verbe contient un attribut qui indique si le verbe dont il dépend est à la forme négative ou non).

Par ailleurs, on a fait attention à fournir un exemple avec « ne » et un exemple avec « n' » pour inférer un motif de transformation suffisamment général.

Motif inféré

```
subtree(v13646,node(6,anything.subtree(1,anything).node(3,tree(neg_ne_ou_vide).subtree(1
,replace(avec_ne,tree(sans_ne))).replace(v21272,tree(nil)).nil).subtree(1,anything).subt
ree(1,tree(pronom_sujet_postverbal_ou_vide)).node(3,tree(neg_pas_ou_vide).subtree(1,repl
ace(avec_pas,tree(sans_pas))).replace(neg_pas.nil.(pas.nil).nil,tree(nil)).nil).nil))
```

Négation d'un verbe avec complément

Exemples fournis pour l'inférence

« l'homme mange les pommes. » → « l'homme ne mange pas les pommes. »

« une femme qui apprécie un enfant applaudit. »

→ « une femme qui n'apprécie pas un enfant applaudit. »

Test concluant

« la fille cohabite-t-elle avec Max? » → « la fille ne cohabite-t-elle pas avec Max? »

Motif inféré

```
subtree(v17074,node(7,tree(groupe_verbal).subtree(1,anything).node(3,tree(neg_ne_ou_vide
).subtree(1,replace(avec_ne,tree(sans_ne))).replace(v26155,tree(nil)).nil).subtree(1,tre
e(verbe_1)).subtree(1,tree(pronom_sujet_postverbal_ou_vide)).node(3,tree(neg_pas_ou_vide
).subtree(1,replace(avec_pas,tree(sans_pas))).replace(neg_pas.nil.
(pas.nil).nil,tree(nil)).nil).subtree(2,replace(avec_ne,tree(sans_ne))).nil))
```

Pronominalisation du sujet

Exemples fournis pour l'inférence

« Max applaudit. » → « il applaudit. »

« une femme par laquelle le chat est attrapé dort. »

→ « une femme par laquelle il est attrapé dort. »

Test concluant

« l'enfant qui chante ne parle pas à Eve. » → « il ne parle pas à Eve. »

Motif inféré

```
subtree(v25228,replace(groupe_nominal.
((sujet.mas.sin).3.sans_determinant.pronom_personnel.sans_chaque.sans_neg.v19419.nil).
(pronom_personnel_sujet.((sujet.mas.sin).3.preverbal.nil).
(il.nil).nil).nil,node(v28216,tree(groupe_nominal).node(7,tree(sujet.mas.sin).tree(3).an
ything.anything.tree(sans_chaque).tree( sans_neg).tree(v16523).nil).node(v31285,tree((su
jet.mas.sin).nil).nil).node(3,anything.node(v30625,tree(sujet.mas.sin).nil).node(v30346,
nil).nil).nil)))
```

Inversion du sujet et du complément (noms propres)

Note

Ceci n'est pas une vraie transformation au sens linguistique telles que les définiraient Chomsky ou Harris. Cependant, elle rentre dans le cadre de notre étude puisqu'elle satisfait à la définition que nous avons donné initialement. C'est elle qui nous a permis de tester le fonctionnement du copier-coller.

Exemples fournis pour l'inférence

« Max aime-t-il Marie? » → « Marie aime-t-elle Max? »

« Max applaudit-il Marie? » → « Marie applaudit-elle Max? »

« Eve parle à Paul. » → « Paul parle à Eve. »

Tests concluants

« Marie connaît-elle Eve? » → « Eve connaît-elle Marie? »

« Eve cohabite avec Claude. » → « Claude cohabite avec Eve. »

Note

Sans le deuxième exemple, on n'aurait pas été capable de réussir le premier test.

Test imparfait

« Max parle à Bob. » se transforme en « Bob parle à Max. » mais aussi en « Bob parle de Max. »

En fait, le motif ne retient pas la préposition du complément et c'est Illico qui complète. Or, dans le cas du verbe parler, deux prépositions conviennent.

Motif inféré

```
subtree(v30506,node(4,tree(proposition).subtree(1,replace(v48886,anything)).node(4,tree(
groupe_nominal).subtree(1,replace(v48886,anything)).subtree(2,replace(v48886,anything)).
node(3,tree(nom_propre).subtree(1,replace(v48886,anything)).subtree(1,replace(v41515,rem
ember(v41342,anything))).nil).nil).node(7,tree(groupe_verbal).subtree(1,replace(v48886,a
nything)).tree(neg_ne_ou_vide.
(sans_ne.nil).nil.nil).subtree(1,node(7,tree(indicatif).replace(v48886,anything).tree(3)
.tree(avoir).anything.replace(v42126,anything).anything.nil)).node(v46335,tree(pronom_su
jet_postverbal_ou_vide).subtree(1,replace(v48886,anything)).replace(v47716,node(v47910,n
il)).nil).tree(neg_pas_ou_vide.
(sans_pas.nil).nil.nil).node(4,tree(groupe_nominal).subtree(1,replace(v42126,anything)).
node(3,tree(preposition_ou_vide).subtree(1,replace(v42126,anything)).replace(v43872,node
(v44066,nil)).nil).node(3,tree(nom_propre).subtree(1,replace(v42126,anything)).subtree(1
,replace(v41342,remember(v41515,anything))).nil).nil).nil).nil))
```

Bibliographie

- Akutsu T., *Tree Edit Distance Problems: Algorithms and Applications to Bioinformatics* , IEICE TRANS. INF. & SYST., Vol. E93–D, No. 2 February 2010 .
- Chomsky N., *The Logical Structure of linguistic Theory*, Plenum, New York, 1975.
- Colmerauer A., *Les grammaires de Métamorphoses*, Rapport de recherche, Groupe Intelligence Artificielle (GIA), Université Aix Marseille II, 1975. Publié en anglais sous le titre *Metamorphosis Grammars* in Natural Language Communication With Computers, Bolc L. (ed), Springer-Verlag, 1978, 133-189.
- Harris Z., *Discourse analysis*, Language 28, No 1 (1952), 1-30.
- Harris Z., *Co-occurrence and transformation in linguistic structure*, Presidential address, Linguistic Society of America, 1955. Language 33, No. 3 (1957), 283-340.
- Hays D., *Grouping and dependency theories*, Proceedings of the National Symposium on Machine Translation, UCLA February 1960 , 258-266.
- Mel'čuk I., *Paraphrase et lexique dans la théorie sens-texte*, *Vingt ans après*, Cahiers de Lexicologie LII, 1988-1.
- Mel'čuk I., *Lexical functions: a tool for the description of lexical relations in the lexicon.* in: Wanner L. (ed.): Lexical functions in lexicography and natural language processing 37–102. John Benjamins, Amsterdam/Philadelphia 1996.
- Mel'čuk I., Wanner L., *Towards a Lexicographic Approach to Lexical Transfer in Machine Translation (Illustrated by the German-Russian Language Pair)*, in Machine Translation, 16, 2001, 21-87 (voir en particulier, 41-43).
- Pasero R., Sabatier P., *ILLICO : Présentation & Principes*, *Connaissances et Formalismes* & *Guide d'utilisation*, Rapport de recherche, [http://www.lif.univ-mrs.fr/illico.html], Laboratoire d'Informatique Fondamentale (LIF), Marseille, 2007.
- Pasero R., Sabatier P., *GNF : une grammaire noyau du français*, Rapport de recherche, [http://www.lif.univ-mrs.fr/illico.html], Laboratoire d'Informatique Fondamentale (LIF), Marseille, 2010.
- Pasero R., Sabatier P., *Contraintes sur les expressions à analyser/produire, principes et applications*, article en préparation, Laboratoire d'Informatique Fondamentale (LIF), 2012.
- Turing A., *Computing Machinery and Intelligence*, in Mind, LIX, No 236, October 1950, 433–460.