Inférence de type et langages à objets à classes

Pierre Crescenzo
Pierre.Crescenzo@unice.fr

Jean-Louis Paquelin
paquelin@i3s.unice.fr

Laboratoire I3S (UNSA-CNRS)
2000, route des lucioles
Les Algorithmes, bâtiment Euclide B
BP 121
F-06903 Sophia Antipolis CEDEX (France)

mots-clefs : Traitement Automatique des Langues Naturelles, Génie Logiciel, langage à objets à classes, inférence de type

1 Introduction

Parfois, on veut réaliser un programme qui doit traiter ses données sans connaître leur type *a priori*, cette information s'affinant au fur et à mesure de l'exécution.

Par exemple, nous souhaitons découvrir l'espèce (le type) d'une plante (l'objet) par son observation. On remarque que cette plante possède des feuilles (un attribut) vertes (la valeur de cet attribut) : on en déduit qu'elle est chlorophyllienne (sous-type de plante). On constate ensuite qu'elle porte des fleurs, c'est donc une phanérogame (sous-type de chlorophyllienne) et ainsi de suite. À la fin de l'analyse, on connaît :

- l'espèce précise de la plante : cela signifie que l'on a, sans ambiguïté, identifié la plante ou
- un ensemble d'espèces possibles : l'observation n'a pas permis d'être plus précis, ou encore,
- une famille dont aucune espèce ne correspond à la plante étudiée : il s'agit soit d'une erreur d'observation, soit de la découverte d'une nouvelle espèce.

Cet exemple est volontairement général et simpliste mais il permet d'illustrer notre but qui est de réaliser une inférence de type dans le cadre d'un langage à objets à classes, tel Java, C++ ou Eiffel. La suite de cet article s'inscrit dans le domaine du *Génie Linguistique* que nous avons introduit par

le passé comme une application des techniques du Génie Logiciel au Traitement Automatique des Langues Naturelles (TALN).

Nous présenterons d'abord l'inférence de type dans le TALN puis les fondements des langages à objets à classes, qui forment un pilier du Génie Logiciel. Nous proposerons ensuite une solution basée sur un système de contraintes appliquées aux types des objets. Nous terminerons en présentant les apports de cette approche et les perspectives de cette étude.

2 Inférence de type dans le TALN

Lorsqu'on s'intéresse à l'implantation d'algorithmes d'analyse (ceux pour lesquels la nature des données est au départ non ou mal connue), on est immédiatement confronté au choix d'un langage de programmation. Toutefois, que l'on ait choisi un langage typé ou non, la notion de type est incontournable. Même dans les langages non typés, elle est en effet présente dans l'esprit du programmeur. Par exemple, en Lisp, célèbre représentant des langages non typés, une liste est une paire mais l'inverse est faux.

En TALN, lorsque l'on souhaite réaliser un analyseur, une tâche essentielle du programme est de déterminer la catégorie grammaticale des mots du texte analysé. Ce travail est très simple lorsqu'on conçoit un analyseur jouet : il est, par exemple, programmé par la consultation d'une table associant chaque mot à sa catégorie. Dans le cas d'un vrai analyseur, il devient indispensable de tenir compte du fait qu'un mot (plus précisément un graphème) peut être catégorisé de différentes façons, cette opération étant un processus progressif d'inférence basé sur les informations glanées au cours de l'analyse.

Dans l'exemple de la figure 1, l'analyse de la phrase « `il ferme sa boutique` »[1] illustre le traitement d'une ambiguïté de niveau lexical. Cette analyse se déroule en deux phases distinctes : la catégorisation qui associe à chaque mot sa catégorie grammaticale (Est-ce un nom, un adjectif, un adverbe, ... ?) puis la composition syntagmatique[2] qui construit les différents ensembles de mots structurant la phrase.

La première étape permet de déterminer sans doute possible la catégorie de trois des quatre mots : `il` est un pronom, `sa` est un adjectif et `boutique` est un nom. Par contre, le second mot de la phrase (`ferme`) ne peut être catégorisé sans ambiguïté : nous disons donc que c'est un mot (et nous pouvons préciser qu'il peut être un adjectif, un adverbe, un verbe ou un nom et rien d'autre).

[1] Pour les non linguistes, précisons que les majuscules et la ponctuation n'ont pas d'importance grammaticale.

[2] Pour les non linguistes, un syntagme est un groupe grammatical (exemple : un syntagme nominal est un groupe nominal).

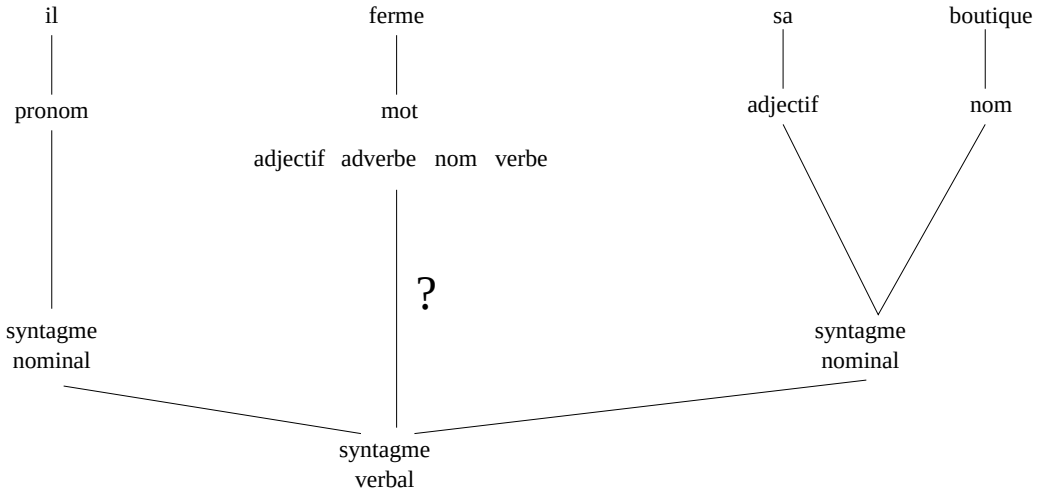

FIG. 1: Exemple d'ambiguïté lexicale

La seconde étape consiste à considérer les syntagmes. Pour simplifier, nous dirons qu'une phrase est constituée (par exemple) d'un syntagme verbal lui-même composé d'un syntagme nominal sujet, d'un verbe puis d'un syntagme nominal complément. Pour que la phrase `il ferme sa boutique` soit grammaticalement correcte, il faut donc que `ferme` soit un verbe.

Nous voyons donc qu'au cours de l'analyse, le *genre* de l'entité `ferme` se précise (mot puis verbe) grâce aux informations recueillies de façon similaire à l'exemple précédent.

3 Langage à objets à classes

Cette section s'adresse principalement aux lecteurs qui ne sont pas familiers avec la terminologie généralement utilisée dans le domaine des langages à objets à classes. Nous allons donc, dans le but de préciser certains termes employés par la suite, donner ici un aperçu de ce domaine.

L'idée de base dans les technologies informatiques dites *à objets* ou *orientées objet* est de représenter les entités physiques ou abstraites du monde réel sous forme d'entités informatiques (on appelle cette opération la *réification*) appelées *objets* décrivant leurs caractéristiques. Ces caractéristiques ou *primitives*, sont représentées d'une part, par des variables d'état de l'objet (appelées *attributs*) et d'autre part, par des comportements de l'objet que l'on nomme *routines*.

La structure (ensemble des routines) de chaque objet est décrite par un *type*. Chaque type peut ainsi modéliser un ensemble d'objets possédant tous la même description et ne différant entre eux que par la valeur de leurs attributs. Chaque type est lui-même décrit par le programmeur sous la forme d'une *classe* qui est une représentation textuelle écrite dans un langages à objets. Une classe est généralement associée à un type et un seul. On nomme *encapsulation* le fait qu'une classe renferme et serve d'interface d'accès à l'ensemble des primitives de ses *instances* (objets *réalisant* la classe par une valuation des attributs). La *migration* est l'opération, souvent non triviale, qui consiste à changer le type d'un objet.

Deux sortes de relations peuvent être établies entre les classes : l'*utilisation* et l'*importation*.

L'utilisation d'une classe par une autre représente la capacité d'un objet à en référencer (*composition par référence*) ou en contenir (*composition par valeur*) un autre. Elle est réalisée au moyen des attributs. L'importation est une relation entre deux classes dont l'une est définie en fonction de l'autre (par ajout, modification ou suppression de primitives). On appelle *référencés*, les objets accessibles par l'intermédiaire des attributs d'un autre objet nommé alors *référençant*.

Le lien d'importation le plus couramment utilisé est nommé *héritage*. Bien que ce ne soit pas son seul usage possible, on en use généralement pour spécifier une relation de *sous-typage*, la modification de primitive étant alors contrainte et la suppression interdite. Enfin, un type est dit *compatible* avec un autre si et seulement s'il en est un sous-type.

4 Une solution

Le processus que nous allons décrire consiste à considérer qu'au départ l'objet à analyser est du type le plus général, racine de l'arbre d'héritage. On souhaite au fur et à mesure de l'obtention d'informations sur cet objet, le déplacer le plus bas possible dans l'arbre. Il s'agit donc en fait, d'un problème de migration, c'est-à-dire d'un changement de type, vers un type plus spécifique.

Pour l'exemple donné en introduction, la racine de l'arbre d'héritage est le type plante, la migration s'effectuant au cours de l'analyse vers les feuille de cet arbre qui représentent les espèces. Les nœuds intermédiaires décrivent les groupes, genres, familles, etc. de la classification botanique. Dans le second exemple (section 1), ...

4.1 Le système de type

Une constante des langages à objets à classes est que l'on associe un type à chaque objet. Dans notre système, nous définissons plutôt qu'un type pour chaque objet, son *domaine typologique* qui est constitué d'un arbre des types possibles pour l'objet.

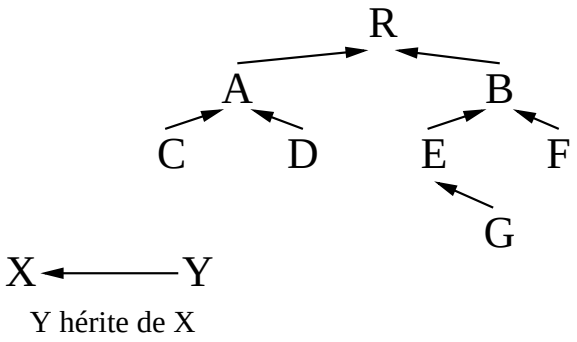

FIG. 2: Domaine typologique initial de x

Prenons immédiatement un exemple pour éclairer notre propos. La figure 2 montre un arbre d'héritage décrivant les différentes types (R, A, B, C, D, E, F et G) d'une application. Disons que nous créons un objet x sans rien spécifier de son type. Par défaut, son domaine typologique sera l'arbre d'héritage complet. On spécifie ensuite (grâce à des informations nouvelles sur l'objet et par un moyen que nous verrons en section 4.2) que cet objet ne peut être de type E. Son nouveau domaine typologique est calculé et donné sur la figure 3. La racine du domaine typologique (R) est appelée *type principal*. Elle décrit en effet le type le plus spécifique qu'on a pu déterminer avec certitude, à ce moment de l'exécution, pour x.

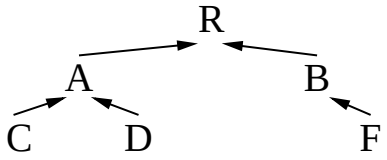

FIG. 3: Domaine typologique de x privé de E

1. Lorsque, à la fin de l'application, le domaine typologique est un arbre réduit à un seul type, cela signifie que l'on a pu déterminer précisément quel est la nature de l'objet analysé.

2. Quand le domaine typologique est un arbre non vide, non réduit à un seul type, cela signifie au contraire que les informations collectées au cours de l'exécution n'ont pas permis de préciser la nature de l'objet. On peut néanmoins utiliser le domaine typologique déduit en énumérant, par exemple, les types possibles.
3. Enfin, si le domaine typologique est vide, cela dénote soit qu'une des informations collectées est fausse, soit que la modélisation du domaine par l'application est incomplète (*i. e.* il manque un type dans l'arbre d'héritage). Ces deux cas de figure, comme les autres, peuvent être exploités par l'utilisateur.

Quel que soit le résultat, il faut garder à l'esprit qu'à l'issue de l'exécution, le domaine typologique contient toute l'information déduite de l'analyse. Par conséquent, plus l'information disponible est précise, plus le domaine typologique le sera.

Au fur et à mesure de l'analyse, le domaine typologique a tendance à se réduire et le type principal à se spécialiser. Il s'agit donc d'un cas particulier de migration : le changement du type de l'objet vers un type héritier.

4.2 Migration pilotée par les contraintes

Dans un langage à objets à classes, on a l'habitude d'écrire des instructions telles x.p = 3 qui décrit une affectation de la valeur 3 dans l'attribut p de l'objet x. De cette instruction, nous pouvons déduire deux choses :

1. x possède un attribut p et
2. cet attribut est de type INTEGER.

Nous noterons que dans cette affectation, c'est le type de 3 qui nous permet une déduction et non sa valeur.

Nous avons choisi de lire cette instruction comme une conjonction des deux contraintes hasPrimitive(x, p) et hasDomain(x.p, INTEGER). Abordons donc maintenant ce langage de contrainte.

On pourra poser plusieurs sortes de contraintes :

- **hasDomain(o, D)** indique que l'objet o a pour domaine typologique D. Cela signifie que :
  1. Le nouveau domaine typologique de o est l'intersection du domaine typologique actuel de o et de D.
  2. Le domaine typologique de tous les objets référencés par les primitives du nouveau o (en fonction de la description donnée par son nouveau type principal) est recalculé.

3. Le domaine typologique de tous les objets référençant o est recalculé.

L'algorithme de hasDomain(o, D) est :

```
début
  soit D' l'intersection de D et du domaine typologique de o
  si D' est vide
  alors
    erreur
  sinon
    soit T le type principal de o
    D' devient le domaine typologique de o
    si le type principal de o est différent de T
    alors
      pour toute primitive p de o ou du type principal de o
        hasPrimitive(o, p)
      finpour
      pour tout o' parmi les référençants de o
        soit p' la primitive référençant o dans o'
        hasPrimitive(o', p')
      finpour
    finsi
  finsi
fin
```

– **hasPrimitive(o, p)** indique que l'objet o possède la primitive p. Cela signifie que :

1. Si o ne possède pas p et qu'il y a au moins un p dans le domaine typologique actuel de o, on le lui ajoute.
2. Le nouveau domaine typologique de o sera le « plus petit arbre possible contenant tous les types du domaine typologique actuel de o qui possèdent une primitive p d'un domaine typologique compatible avec le domaine typologique de l'objet référencé par p dans l'actuel o ». Propagation des changements aux référencés et référençants.
3. Le nouveau domaine typologique de l'objet référencé par p dans o sera le « plus petit arbre possible contenant tous les domaines typologiques de p dans tous les types (qui contiennent p) du nouveau domaine typologique de o ». Propagation des changements aux référencés et référençants.

L'algorithme de hasPrimitive(o, p) est :

```
début
  soit E l'ensemble des types du domaine typologique de o qui possèdent une primitive p d'un domaine typologique compatible avec le domaine typologique de l'objet référencé par p dans o[3]
  si E est vide
  alors
    erreur
  sinon
    si o ne possède pas p
    alors
      ajoute p à o
    finsi
    soit D le domaine typologique formé du plus petit arbre possible contenant tous les types de E
    si D est différent du domaine typologique de o
    alors
      hasDomain(o, D)
      soit E' l'ensemble des domaines typologiques de p dans tous les types (qui contiennent p) de D
      soit D' le domaine typologique formé du plus petit arbre possible contenant tous les types de E'
      soit o' l'objet référencé par p dans o
      si D' est différent du domaine typologique de o'
      alors
        hasDomain(o', D')
      finsi
    finsi
  finsi
fin
```

- **hasNotType(o, T)** indique que l'objet o n'est pas du type T, ni d'un de ses descendants. Cela signifie que :
  1. Le nouveau domaine typologique de o est la différence entre domaine typologique actuel de o et le domaine typologique issu de T. Propagation des changements aux référencés et référençants.

  L'algorithme de hasNotType(o, T) est :

[3] Si o ne possède pas p alors la compatibilité est toujours vraie.

```
début
  soit D' le domaine typologique issu de T dans l'arbre d'héritage
  soit D le domaine typologique, résultat de la soustraction de D' au
    domaine typologique de o
  si D est vide
  alors
    erreur
  sinon
    hasDomain(o, D)
  finsi
fin
```

On pourrait s'attendre à disposer d'une contrainte hasNotDomain(o, D), qui indiquerait que o ne peut être d'aucun type du domaine typologique D, plutôt que hasNotType(o, T). Cependant, la hiérarchie de sous-typage exprime qu'un type T1 hérite de toutes les propriétés de son père T0 et donc qu'un objet qui ne peut être du type T0 ne peut pas non plus être du type T1. Par conséquent, hasNotDomain(o, D) ne peut avoir comme sémantique que « o ne peut être du type principal de D, ni d'un de ses descendants » ce qui nous ramène à la sémantique de hasNotType(o, T).

- **hasNotPrimitive(o, p)** indique que l'objet o ne possède pas la primitive p. Cela signifie que :
  1. Le nouveau domaine typologique de o sera le « plus petit arbre possible contenant tous les types du domaine typologique actuel de o qui ne possèdent pas une primitive p ». Propagation des changements aux référencés et référençants.

  L'algorithme de hasNotPrimitive(o, p) est :

```
début
  soit E l'ensemble des types du domaine typologique de o qui pos-
    sèdent la primitive p
  soit D le domaine typologique, résultat de la soustraction de E au
    domaine typologique de o
  si D est vide
  alors
    erreur
  sinon
    hasDomain(o, D)
  finsi
fin
```

Ces contraintes permettent de raffiner le domaine typologique et donc le type des objets analysés. Ce raffinage consistant en une descente dans l'arbre d'héritage (on ne peut jamais remonter ni sauter dans une autre branche), nous avons donc affaire à une migration contrôlée par ces contraintes, la migration ne s'effectuant que vers un type plus spécifique.

4.3 Discussion sur les opérateurs

L'affectation (opérateur =), en tant que brique de base des langages impératifs, va être reconsidérée dans le contexte de l'inférence de type. Il en est de même pour l'opération d'accès (opérateur .) et celle de création. Cependant, même si nous les redéfinissons, ces opérations continuent évidemment d'exister mais seulement au plus bas niveau. En effet, le langage doit être cohérent et ne pas permettre de manipulation des objets en dehors du système d'inférence de type.

La sémantique de la création, de l'accès et de l'affectation dans notre système est donnée dans le tableau ci-dessous.

création de o avec le type T	allocate(o) hasDomain(o, domainOf(T)) initialize(o)
accès à la primitive p de o	hasPrimitive(o, p) o.p
affectation de o' à o	hasDomain(o, domainOf(o')) o = o'

5 Perspectives et conclusion

Notre système permet de résoudre élégamment et dans le cadre du Génie Logiciel le type de problème donné en introduction. Il ne limite en rien les capacités actuelles des langages à objets à classes et il est adapté à certains problèmes du Traitement Automatique des Langues Naturelles, par exemple, le traitement de l'ambiguïté.

Les perspectives de ce travail sont :

- l'intégration des exceptions dans le système de types qui pourrait se faire selon deux approches :
  1. un recalcul de l'arbre de typage afin de replacer automatiquement les types des exceptions pour conserver la cohérence de l'arbre ou
  2. la modification de la sémantique du sous-typage et de la migration.
- l'implantation du système de type présenté sera d'abord réalisé sous la forme d'une extension de Java.