Combining Functionality and Object-Orientedness for Natural Language Processing

Toyoaki Nishida[1] and Shuji Doshita
Department of Information Science, Kyoto University
Sakyo-ku, Kyoto 606, JAPAN

Abstract

This paper proposes a method for organizing linguistic knowledge in both systematic and flexible fashion. We introduce a purely applicative language (PAL) as an intermediate representation and an object-oriented computation mechanism for its interpretation. PAL enables the establishment of a principled and well-constrained method of interaction among lexicon-oriented linguistic modules. The object-oriented computation mechanism provides a flexible means of abstracting modules and sharing common knowledge.

1. Introduction

The goal of this paper is to elaborate a domain-independent way of organizing linguistic knowledge, as a step forwards a cognitive processor consisting of two components: a linguistic component and a memory component.

In this paper we assume the existence of the latter component meeting the requirements described in [Schank 82]. Thus the memory component attempts to understand the input in terms of its empirical knowledge, predict what happens next, and reorganize its knowledge based on new observations. Additionally, we assume that the memory component can judge whether a given observation is plausible or not, by consulting its empirical knowledge.

The role of the linguistic component, on the other hand, is to supply "stimulus" to the memory component. More specifically, the linguistic component attempts to determine the propositional content, to supply missing constituents for elliptical expressions, to resolve references, to identify the focus, to infer the intention of the speaker, etc. In short, the role of the liguistic component is to "translate" the input into an internal representation.

For example, the output of the linguistic component for an input:

When did you go to New York?

is something like the following[2]:

> There is an event e specified by a set of predicates: isa(e)=going $\wedge$ past(e) $\wedge$ agent(e)=the_hearer $\wedge$ destination(e)=New_York. The speaker is asking the hearer for the time when an event e took place. The hearer presupposes that the event e actually took place at some time in the past.

[1]Currently visiting Department of Computer Science, Yale University, New Haven, Connecticut 06520, USA.

If the presupposition contradicts what the memory component knows, then the memory component will recognize the input as a loaded question [Kaplan 82]. As a result, the memory component may change its content or execute a plan to informing the user that the input is inconsistent with what it knows.

The primary concern of this paper is with the linguistic component. The approach we take in this paper is to combine the notion of compositionality[3] and an object-oriented computational mechanism to explore a principled and flexible way of organizing linguistic knowledge.

2. Intermediate Representation and Computational Device for Interpretation

2.1 PAL (Purely Applicative Language)

Effective use of intermediate representations is useful. We propose the use of a language which we call PAL (Purely Applicative Language).

In PAL, new composite expressions are constructed only with a binary form of function application. Thus, if x and y are well-formed formulas of PAL, so is a form $x(y)$. Expressions of PAL are related to expressions of natural language as follows:

Generally, when a phrase consists of its immediate descendants, say x and y, a PAL expression for the phrase is one of the following forms:

$$<x>(<y>) \text{ or } <y>(<x>)$$

where $<\alpha>$ stands for a PAL expression for a phrase α. Which expression is the case depends on which phrase modifies which. If a phrase x modifies y then the PAL expression for x takes the functor position, *i.e.*, the form is $<x>(<y>)$.

Simple examples are:

big apple $\Rightarrow$ big(apple) *; adjectives modify nouns*
very big $\Rightarrow$ very(big) *; adverbs modify adjectives*
very big apple $\Rightarrow$ (very(big))(apple) *; recursive composition*

[2]As illustrated in this example, we assume a predicate notation as an output of the linguistic component. But this choice is only for descriptive purposes and is not significant.

[3]We prefer the term "functionality" to "compositionality", reflecting a procedural view rather than a purely mathematical view.

How about other cases? In principle, this work is based on Montague's observations [Montague 74]. Thus we take the position that noun phrases modify (are functions of, to be more precise) verb phrases. But unlike Montague grammar we do not use lambda expressions to bind case elements. Instead we use special functors standing for case markers. For example,

he runs ⇒ (*subject(he))(runs)
he eats it ⇒ (*subject(he))((*object(it))(eats))

Another example, involving a determiner, is illustrated below:

a big apple ⇒ a(big(apple)) *; determiners modify nouns*

Sometimes we assume "null" words or items corresponding to morphemes, such as, role indicators, nominalizer, null NP, *etc.*

```
apple which he eats
          ⇒ (which
              ((*subject(he))
                 ((*object(*null))
                    (eats))))
            (apple)
    ; restrictive relative clauses modify nouns,
    ; relativizers modify sentences to make adjectives
```

In the discussion above, the notion of *modify* is crucial. What do we mean when we say *x modifies y*? In the case of Montague grammar, this question is answered based on a predetermined set theoretical model. For example, a noun is interpreted as a set of entities; the noun "penguin", for instance, is interpreted as a set of all penguins. An adjective, on the other hand, is interpreted as a function from sets of entities to sets of entities; an adjective "small" is interpreted as a selector function which takes such a set of entities (interpretation of each noun) and picks up from it a set of "small" entities. Note that this is a simplified discussion; intension is neglected. Note also that different conception may lead to a different definition of the relation *modify*, which will in turn lead to intermediate representations with different function-argument relationships.

After all, the choice of semantic representation is relative to the underlying model and how it is interpreted. A good choice of a semantic representation - interpretation pair leads to a less complicated system and makes it easier to realize.

The next section discusses a computational device for interpreting PAL expressions.

2.2 Object-Oriented Domain

The notion of object-orientedness is widely used in computer science. We employ the notion in LOOPS [Bobrow 81]. The general idea is as follows:

> We have a number of objects. Objects can be viewed as both data and procedures. They are data in the sense that they have a place (called a local variable) to store information. At the same time, they are procedures in that they can manipulate data. An object can only update local variables belonging to itself. When data belongs to another object, a message must be sent to request the update. A message consists of a label and its value. In order to send a message, the agent has to know the name of the receiver. There is no other means for manipulating data. Objects can be classified into classes and instances. A class defines a procedure (called a method) for handling incoming messages of its instances. A class inherits methods of its superclasses.

3. Interpretation of PAL Expressions in Object-Oriented Domain

A class is defined for each constant of PAL. A class object for a lexical item contains linguistic knowledge in a procedural form. In other words, a class contains information as to how a corresponding lexical item is mapped into memory structures.

A PAL expression is interpreted by evaluating the form which results from replacing each constant of a given PAL expression by an instance of an object whose class name is the same as the label of the constant. The evaluation is done by repeating the following cycle:

- an object in argument position sends to an object in functor position a message whose label is "argument" and whose value is the object itself.
- a corresponding method is invoked and an object is returned as a result of application; usually one object causes another object to modify its content and the result is a modified version of either a functor or an argument.

Note that objects can interact only in a constrained way. This is a stronger claim than that allowing arbitrary communication. The more principled and constrained way modules of the linguistic component interact, the less complicated will be the system and therefore the better perspective we can obtain for writing a large grammar.

3.1 A Simple Example

Let's start by seeing how our simple example for a sentence "he runs" is interpreted in our framework. A PAL expression for this sentence is:

(*subject(he))(runs)

Class definitions for related objects are shown in figure 3.1.

The interpretation process goes as follows:

- Instantiating '*subject': let's call the new instance $*subject_0$.
- Instantiating 'he': a referent is looked for from the memory. The referent (let's call this i_0) is set to the local variable *den*, which stands for 'denotation'. Let the new instance be he_0.
- Evaluating '$*subject_0(he_0)$': a message whose label is 'case' and whose value is 'subject' is sent to the object he_0. As a result, he_0's variable *case* has a value 'subject'. The value of the evaluation is a modified version of he_0, which we call he_1 to indicate a different version.
- Instantiating 'runs': let's call the new instance $runs_0$. An event node (of the memory component) is created and its reference (let's call this e_0) is set to the local variable *den*. Then a new proposition '$takes\_place(e_0)$' is asserted to the memory component.

```
class *subject:
  argument: send[message, case:subject];
            return[self].
    ; if a message with label 'argument' comes, this method will send to
      the object pointed to by the variable message a message whose
      label is 'case' and whose value is 'subject'.
    ; a variable message holds the value of an incoming message and a
      variable self points to the object itself.

class he:
  if instantiated then den←'look_for_referent'.
    ; when a new instance is created, the referent is looked for and the
      value is set to the local variable den.
  case: case←message; return[self].
    ; when a message comes which is labeled 'case', the local variable case
      will be assigned the value the incoming message contains. The
      value of this method is the object itself.
  argument: return[send[message, case:self]];
    ; when this instance is applied to another object, this object will send
      a message whose label is the value of the local variable case and
      whose value field is the object itself. The value of the message
      processing is the value of this application.

class runs:
  if instantiated then den←create['event:run'];
                       assert[takes_place(den)].
    ; when a new instance of class 'runs' is instantiated, a new event will
      be asserted to the memory component. The reference to the new
      event is set to the local variable den.
  subject: assert['agent(den)=message.den']; return[self].
    ; when a message with label 'subject' comes, a new proposition is
      asserted to the memory component. The value of this message
      handling is this object itself.
```

Figure 3-1: Definitions of Sample Objects

- Evaluating $he_1(runs_0)$: a message whose label is 'subject' and whose value is he_1 is sent to $runs_0$, which causes a new proposition 'agent(e_0)=i_0' to be asserted in the memory component. The final result of the evaluation is a new version of the object $runs_0$, say $runs_1$.

The above discussion is overly simplified for the purpose of explanation. The following sections discuss a number of other issues.

3.2 Sharing Common Knowledge

Object-oriented systems use the notion of hierarchy to share common procedures. Lexical items with similar characterics can be grouped together as a class; we may, for example, have a class 'noun' as a superclass of lexical items 'boy', 'girl', 'computer' and so forth. When a difference is recognized among objects of a class, the class may be subdivided; we may subcategorize a verb into static verbs, action verbs, achievement verbs, *etc.* Common properties can be shared at the superclass. This offers a flexible way for writing a large grammar; one may start by defining both most general classes and least general classes. The more observations are obtained, the richer will be the class-superclass network. Additionally, mechanisms for supporting a multiple hierarchy and for borrowing a method are useful in coping with sophistication of linguistic knowledge, *e.g.*, introduction of more than one subcategorization.

3.3 Linking Case Elements

One of the basic tasks of the linguistic component is to find out which constituent is linked explicitly or implicitly to which constituent. From the example shown in section 3.1, the reader can see at least three possibilities:

Case linking by sending messages. Using conventional terms of case grammar, we can say that "governer" receives a message whose label is a surface case and whose value is the "dependant". This implementation leads us to the notion of abstraction to be discussed in section 3.4.

Lexicon-driven methods of determining deep case. Surface case is converted into deep case by a method defined for each governer. This makes it possible to handle this hard problem without being concerned with how many different meanings each function word has. Governers which have the same characteristics in this respect can be grouped together as a superclass. This enables to avoid duplication of knowledge by means of hierarchy. The latter issue is discussed in section 3.2.

The use of implicit case markers. We call items such as *subject or *object implicit, as they do not appear in the surface form, as opposed to prepositions, which are explicit (surface) markers. The introduction of implicit case marker seems to be reasonable if we see a language like Japanese in which surface case is explicitly indicated by postpositions. Thus we can assign to the translation of our sample sentence a PAL expression with the same structure as its English version:

KARE GA HASHIRU ⇒ (GA(KARE))(HASHIRU)

where, "KARE" means "he", "GA" postposition indicating surface subject, "HASHIRU" "run", respectively.

3.4 Abstraction

By attaching a sort of a message controller in front of an object, we can have a new version of the object whose linguistic knowledge is essentially the same as the original one but whose input/output specification is different. As a typical example we can show how a passivizer *en is dealt with. An object *en can have an embedded object as a value of its local variable *embedded*. If an instance of *en receives a message with label '*subject', then it will send to the object pointed by *embedded* the message with its label replaced by '*object'; if it receives a message with label 'by', then it will transfer the message to the "embedded" object by replacing the label field by '*subject'.

Thus the object *en coupled with a transitive verb can be viewed as if they were a single intransitive verb. This offers an abstracted way of handling linguistic objects.

The effect can be seen by tracing how a PAL expression:

```
(*subject(this(sentence)))
   ((by(a(computer)))
      (*en(understand)))
     "This sentence is understood by a computer."
```

is interpreted[4].

[4]Notice how the method for a transitive verb "understand" is defined, by extending the definition for an intransitive verb "run".

3.5 Implicit Case Linking

We can use a very similar mechanism to deal with case linking by causative verbs. Consider the following sentence:

x wants *y* to *do z*.

This sentence implies that the subject of the infinitive is the grammatical object of the main verb "wants". Such a property can be shared by a number of other verbs such as "allow", "cause", "let", "make", *etc.* In the object-oriented implementation, this can be handled by letting the object defined for this class transfer a message from its subject to the infinitive.

Note that the object for the these verbs must pass the message from its subject to the infinitive when its grammatical object is missing.

Another example of implicit case linking can be seen in relative clauses. In an object-oriented implementation, a relativizer transfers a message containing a pointer to the head noun to a null NP occupying the gap in the relative clause. Intermediate objects serve as re-transmitting nodes as in computer networks.

3.6 Obligatory Case *versus* Non-Obligatory Case

In building a practical system, the problem of distinguishing obligatory case and non-obligatory case is always controversial. The notion of hierarchy is useful in dealing with this problem in a "lazy" fashion. What we means by this is as follows:

> In procedural approach, the distinction we make between obligatory and non-obligatory cases seems to be based on economical reason. To put this another way, we do not want to let each lexical item have cases such as locative, instrumental, temporal, *etc.* This would merely mean useless duplication of knowledge. We can use the notion of hierarchy to share methods for these cases. Any exceptional method can be attached to lower level items.
>
> For example, we can define a class "action verb" which has methods for instrumental cases, while its superclass "verb" may not.

This is useful for not only reflecting linguistic generalization but also offering a grammar designer a flexible means for designing a knowledge base.

4. A Few Remarks

As is often pointed out, there are a lot of relationships which can be determined purely by examining linguistic structure. For example, presupposition, intra-sentential reference, focus, surface speech acts, *etc.* This eventually means that the linguistic component itself is domain independent.

However, other issues such as, resolving ambiguity, resolving task-dependent reference, filling task-dependent ellipsis, or inferring the speaker's intention, cannot be solved solely by the linguistic component [Schank 80]. They require interaction with the memory component. Thus the domain dependent information must be stored in the memory component.

To go beyond the semantics-on-top-of-syntax paradigm, we must allow rich interaction between the memory and linguistic components. In particular, the memory component must be able to predict a structure, to guide the parsing process, or to give a low rating to a partial structure which is not plausible based on the experience, while the linguistic component must be able to explain what is going on and what it tries to see. To do this, the notion of object-orientedness provides a fairly flexible method of interaction.

Finally, we would like to mention how this framework differs from the authors' previous work on machine translation [Nishida 83], which could be viewed as an instantiation of this framework. The difference is that in the previous work, the notion of lambda binding is used for linking cases. We directly used intensional logic of Montague grammar as an intermediate language. Though it brought some advantages, this scheme caused a number of technical problems. First, using lambda forms causes difficulty in procedural interpretation. In the case of Montague grammar this is not so, because the amount of computation doet not cause any theoretical problem in a mathematical theory. Second, though lambda expressions give an explicit form of representing some linguistic relations, other relations remain implicit. Some sort of additional mechanism should be introduced to cope with those implicit relations. Such a mechanism, however, may spoil the clarity or explicitness of lambda forms. This paper has proposed an alternative to address these problems.

Acknowledgements

We appreciate the useful comments made by Margot Flowers and Lawrence Birnbaum of Yale University, Department of Computer Science.

References

[Bobrow 81] Bobrow, D. G. and Stefik, M.
The LOOPS Manual.
Technical Report KB-VLSI-81-13, Xerox PARC, 1981.

[Kaplan 82] Kaplan, S. J.
Cooperative Responses from a Portable Natural Language Query System.
Artificial Intelligence 19(1982):165-187, 1982.

[Montague 74] Montague, R.
Proper Treatment of Quantification in Ordinary English.
In Thompson (editor), *Formal Philosophy*, pages 247-270. Yale University, 1974.

[Nishida 83] Nishida, T.
Studies on the Application of Formal Semantics to English-Japanese Machine Translation.
Doctoral Thesis, Kyoto University, 1983.

[Schank 80] Schank, R. C. and Birnbaum.
Memory, Meaning, and Syntax.
Technical Report 189, Yale University, Department of Computer Science, 1980.

[Schank 82] Schank, R. C.
Dynamic Memory: A Theory of Reminding and Learning in Computers and People.
Cambridge University Press, 1982.