Natural Language Processing with Hypothetical Assumptions, Hidden Accumulator Grammars and Sparse Term Taxonomies

Paul Tarau
Université de Moncton
Moncton, Canada, E1A 3E9
`tarau@info.umoncton.ca`

Veronica Dahl
Logic Programming Group and
Computing Sciences Department
Simon Fraser University
`veronica@cs.sfu.ca`

Andrew Fall
Logic Programming Group and
Computing Sciences Department
Simon Fraser University
`fall@cs.sfu.ca`

Abstract

We introduce a framework for natural language processing based on linear and intuitionistic assumptions, a new type of logic grammars (HAGs), and a sparse term encoding of sorts used to implement semantic taxonomies.

Our framework is able to infer type relationships as well as deal with backtrackable state information in a principled yet highly efficient way.

Intuitionistic and *linear* implications scoped over the current continuation allow us to follow given branches of the computation under hypotheses that disappear when and if backtracking takes place. Unlike previous similar frameworks, ours maintains high-level descriptiveness through the use of hidden *multiple accumulators.* This new technique offers the equivalent of EDCGs [Van89] without the need for a preprocessor.

As a 'proof-of-concept', a 'world-modeling' program is built by *abstracting* from natural language sentences, hypothetical assumptions which are later projected back to a 'concrete domain' to handle anaphora and simple common sense knowledge problems.

A type hierarchy of concepts is automatically constructed and efficiently encoded using *sparse terms* from basic definitions given by the user, and can be consulted, also in high level ways, in order to draw inferences re. class belongings and inherited properties.

Keywords: natural language processing, logic grammars, linear (intuitionistic) implication, hypothetical assumptions, alternative DCG implementations, Hidden Accumulator Grammars, backtrackable destructive assignment, sparse terms, taxonomies.

1 Introduction

The idea of using linear and intuitionistic logic for natural language processing is not new. In [PM90], for instance, a method was proposed for handling relativization by temporarily augmenting a λProlog grammar with a missing noun phrase rule, only during the parse of a relative clause. Thus, in "the review that Amy completed", for instance, the relative clause can be viewed as a sentence in which the object noun phrase required for "completed" is missing (to be equated with the antecedent "the review"). While allowing for empty noun phrases anywhere in the sentence would lead to overgeneration (e.g., sentences with missing subjects) or need to be controlled through cumbersome extra arguments of multiple-headed grammar rules (see e.g. [AD89]), the linear logic approach offers a clean and effective formulation, even though it admittedly has some problems (e.g. it admits erroneous sentences in which a relative clause has no missing noun phrase, as in *"the house that jack built the cottage"). These problems were later addressed by Miller and Hodas in a linear logic framework [Hod92a, HM92] in which a relative clause rule can be expressed as follows (modulo notation changes for standard readability):

```
rel([that|L1],L2):- np(Z,Z) -o s(L1,L2).
```

This rule expresses that a relative clause can be recognized in a string starting with "that" if we can recognize a sentence in the remainder (L1) of the input string, by hypothesizing a missing noun phrase only during the derivation of that sentence. Since the hypotheses must be used, relative clauses with no empty np are correctly avoided.

Two things should be noticed about this rule:

- the input and output strings (L1 and L2) MUST be explicitly stated, thus forcing us to regress to pre-DCG grammar formulations, and
- the missing noun phrase can be anywhere in the relative sentence, so that specifying a particular kind of relativization (on the subject as in "the book that arrived today", on the object as in "the book that I read today", or on a complement as in "the student that I gave a book to"), and the semantic relation of the missing noun phrase with its antecedent, might not be very straightforward to achieve.

In Section 2, after introducing Hidden Accumulator Grammars (HAGs), we shall see how the use of accumulators can simplify this treatment and result in more readable code, while correctly attaching the antecedent to different kinds of possibly missing noun phrases as well. Section 3 describes a restricted 'executable English' program which solves anaphora based on a world modeling approach using a form of abstract interpretation over a semantic domain and hypothetical state transitions. We then show the use of HAGs for describing long distance dependencies in section 4.

The idea of using type information in natural language processing is also not a new one. Within logic programming contexts it has the specific attraction of being representable by incomplete terms, i.e., terms containing logical variables which can be further specified dynamically through unifications, so that operations such as type intersection can reduce to little more than unification. This has been exploited for instance in [Dah91, AKP93]. More recently, *sparse terms* [Fal95] have been developed as a flexible and efficient representation in which incomplete information can be easily extended through unification. Sparse terms have also been studied as means to efficiently encode taxonomies in a form that is amenable to computing operations such as *greatest lower bound* and *subsumption* [Fal94]. Taxonomic encoding automates the management of partially ordered information, and is a fundamental component of ψ-term management in LIFE [AKP93].

Because these hierarchies are invisibly managed, we can view them as another case of hypothetical reasoning (albeit of a different nature than that allowed by linear or intuitionistic implication), in that it provides automatic reasoning on the hypotheses we usually make about type hierarchies and their inheritance properties.

Section 5 exploits type hierarchies as a tool for the difficult linguistic problem of anaphora resolution. In Section 6 we describe a flexible approach to constructing hierarchies using *sparse terms* [Fal95]. Section 7 discusses related work, and section 8 gives some areas for future research.

2 A hypothetical reasoning based framework for NL processing

The basic components of our framework are the following:

1. linear and intuitionistic implications scoped over the current continuation
2. hidden multiple accumulator grammars (HAGs)
3. reasoning on virtual type hierarchies deduced from a user's definitions.

We next shortly describe each of these extensions in turn. We refer to the companion paper [TDF95] for the details of their design and implementation in BinProlog [Tar95].

2.1 Linear and intuitionistic implication

Intuitionistic assumei/1 adds temporarily a clause usable in later proofs. Such a clause can be used an indefinite number of times, like asserted clauses, except that it vanishes on backtracking. Its scoped version `Clause=>Goal` or `[File]=>Goal` makes `Clause` or the set of clauses found in `File` available only during the proof of Goal and corresponds to the intuitionistic implication of λProlog. While assumption A in the intuitionistic implication `A=>B` is usable only within the proof of B, after assumei(A), A is usable within the current AND-continuation.

Linear assumel/1 adds temporarily a clause usable *at most once* in later proofs. This assumption also vanishes on backtracking. Its scoped version `Clause -: Goal`[1] or `[File] -: Goal` makes `Clause` or the set of clauses found in `File` available only during the proof of Goal. Both vanish on backtracking. Linear implication and assumption can be seen as implicitly existentially quantified, i.e. the original copy

[1]The use of `-:` instead of the usual `-o` comes from the fact that in Prolog an operator mixing alphabetic and special characters would require quoting in infix position. Also, since our implication differs semantically from usual linear implication, it seems reasonable to denote it differently.

is used in further unifications, while intuitionistic implication and assumption behave like asserts with the usual copying semantics, both on use and on definition. Although assumed clauses can be called directly, it is good programming practice (and slightly more efficient) to call them through the use of BinProlog's `assumed/1` builtin which will look directly to assumed code without attempting to find first a statically compiled definition.

Programmer enforced weakening rule The *weakening* rule in linear logic requires every assumption to be eventually used. For the natural language world we intend to model, when assumptions range over the current continuation, this requirement seems too strong, except for the well-known situation of handling relatives through the use of gaps [Hod92a].

Therefore, we have left *weakening* in the hand of the programmer who can simply specify (by using for instance, negation as failure) that, at a given point, the set of left over assumptions should be empty.

2.2 NL processing as world-modeling

Realistic natural language analysis cannot make abstraction of semantics and pragmatics as programming languages cannot fully make abstraction of their run-time environments.

Computer-based discourse understanding (as it's human counterpart) is basically a form of model-building. It involves constant constraint-solving to keep, at a given time, only a manageable subset of (intended) models. The task is harder, but similar to that of compilers for programming languages.

Following the compiler analogy, anaphora resolution parallels the process of symbol resolution in linkers. As in the case of dynamic linking, some of this information is provided on the fly, and unsolved references at a given point make sense as far as they will be solved in the future. In this case anaphora and ambiguity are likely to be resolved with an *abductive* world-modeling process. Each sentence of the discourse generates and possibly consumes/solves various hypotheses and constraints. The process is backtrackable although some form of commit can be provided when common sense reasoning allows it or imposes it.

The effect of every sentence on the 'world-model' is described with a set of *pre-* and *post*-conditions which are either *used* or *defined* as facts during the analysis.

Various discourse fragments exhibit two distinct modes: *define* and *use.*

Let us illustrate this with two sentences and a (subset of) their post-conditions.

```
``a dog bit john''

  define: bitten_by(a_dog,john)

``the dog that bit john is smart''

  use: bitten_by(a_dog,john)
  define: smart(a_dog)
```

Suppose a sentence refers to an individual through an anaphora (pronoun, definite article) which is not yet defined in the discourse. It is reasonable to delay resolving the referent and abduce (make the hypothesis) that such an individual exists. Information from the current sentence will be used to generate constraints on the referent of the anaphora. When the anaphora is tentatively solved, abduced constraints are used to eliminate inconsistent interpretations.

A difficult problem is the one of 'dereferencing' a given description. In the general case this points to the (undecidable) problem of testing the identity of two first-order objects. A way to prevent this from the start is to *abstract* identity information, so that this *dereferencing* become trivial. This loss of information, to be tolerable, requires a way to project back the abstract information into the concrete domain, where the referent is additionally constrained by the current state of the world. The next section illustrates these general principles with a 'proof-of-concept' implementation.

3 World-modeling based anaphora resolution

We will show how simple it is to handle anaphora using a set of 'abstracted' hypothetical world-state transitions. Our *abstraction operator* will 'forget' information which would get in the way of proving the equivalence between a given assumption and its referent. Combined with operator parsing, which handles syntax for free (see Appendix I), this give the illusion of 'executable English' on a restricted 'MUD'-style closed world. Transitions on the state of the world are obtained by executing (in sequence) the following 'story'.

```
text(there is a book in the green room).
text(mary has a flower).
text(joe is in the green room).
text(he has a diamond).
text(she is in the blue room).
text(he takes the book).
text(he goes to the blue room).
text(he gives it to her).
text(mary drops the book).
text(he takes it).
text(he goes to the red room).
text(she goes to the red room).
text(she gives a flower to him).
text(
  if joe has the moon
  then joe gives the moon to mary
  else joe gives his diamond to her
).
text(joe goes to the green room).
```

Here is the snapshot for the transition induced by our 13-th sentence:

```
ABSTRACT ASSUMPTIONS BEFORE:

% assumed (has)/2:
mary has flower.
joe has diamond.
joe has book.
red room has joe.
red room has mary.

READING: she gives a flower to him
```

```
PARSING:
                                  gives
                                    |
                             .--------.
                             |        |
                            she      to
                                      |
                                   .-----.
                                   |     |
                                   a    him
                                   |
                                   .
                                   |
                                flower

ABSTRACT ASSUMPTIONS AFTER:

% assumed (has)/2:
joe has diamond.
joe has book.
red room has joe.
red room has mary.
joe has flower.
```

Verbs are described by their action on the MUD. Here is the semantics of *takes*:

```
takes(S0,O0):-
  p_nom(S0,S), % S is a dereferenced nominative semantic root
  p_acc(O0,O), % O is a dereferenced accusative semantic root
  check(P has S),  % constraint: P should have S
  transfer(O,P,S). % O is transfered from P to S
```

A linear assumption based 'non-destructive' check/1 operation can be defined when we do not want the facts to be consumed (see [TDF95] for details):

```
check(X):-X,assumel(X).
```

Note that a verb very different at surface level like *goes* is abstracted to a very similar transition.

```
goes(S0,O0):-
  p_nom(S0,S), % S0 dereferences to S
  p_acc(O0,O), % P0 dereferences to O
  transfer(S,_,O). % S is transfered to O
```

We refer to Appendix I for the (unusually compact) implementation of this 'world-modeling' based natural language processing example. At the abstract level, state is expressed uniformly as assumptions about *"people having things"* and *"places having people"*. Words are abstracted to their minimal *semantic roots* and then *dereferenced* to known individuals [2]. A concretization step is needed in case of a dialog generation component (for *knobots* in the MUD) to put back the missing 'flesh' on the skeletons, before outputting them as surface level sentences.

[2]This can either be delayed by using an abductive reasoning component to the future of the discourse or 'solved' by creating synthetic individuals (skolemization).

3.1 Hidden Accumulators

When we want to avoid any search and enforce some sequentiality on the use of linear assumptions (a stack discipline, for instance) we will use the observation of [TDF95] that accumulator processing (EDCGs) is a special instance of hypothetical reasoning with linear assumptions. BinProlog's Hidden Accumulator Grammars are a variant of EDCGs which do not require preprocessing. They have been implemented in C and are accessible through the following set of built-ins:

```
dcg_connect/1  % works like 'C'/3 with 2 invisible arguments
dcg_def/1   % sets the first invisible DCG argument
dcg_val/1   % retrieves the current state of the DCG stream
dcg_tell/1  % focuses on a given DCG stream
dcg_telling/1 % returns the number of the current DCGs stream
```

On top of them it is possible to define a 'multi-stream' phrase/3 construct,

```
dcg_phrase(DcgStream, Axiom, Phrase)
```

that switches to the appropriate `DcgStream` and uses `Axiom` to process or generate `Phrase`. We refer to the companion paper [TDF95] for their specification in term of linear assumptions.

4 Describing long distance dependencies through HAGs

As mentioned in the Introduction, previous linear logic based approaches to long distance dependencies, such as relativization, force us to go back to explicitly coding the input and output string in every rule.

By using the features of HAGs we can restore high level expressiveness. Consider the following rules, which describe simple sentences relativizing on a verb's subject, object or complement. They construct a three-branched quantification-based semantics.

```
sent(R) :- np(X,VP,R), vp(X,VP).
% build R fro, the subject's representation
% (or quantified variable) X and the verb
% phrase's representation VP

np(X,VP,VP):- proper_name(X).
np(X,VP,R):- det(X,NP,VP,R), noun(X,N1), rel(X,N1,NP).
% build the scope NP fro, the representation N1 of the
% head and that of the relative clause
np(X,VP,VP):- assumed(antecedent(X)). % consume the missing noun phrase's
% antecedent (hypothesized by the second
% rule for rel, and equate it with the
% variable introduced by the missing np
det(X,NP,VP,a(X,NP,VP)) :- {a}.
det(X,NP,VP,the(X,NP,VP)):- {the}.

% build a three-branched quantification from a variable X,
% a scope NP and a restriction VP.
```

```
noun(X,woman(X)):- {woman}.
noun(X,title(X)):- {title}.

proper_name(john):- {john}.

rel(X,X).   % empty relative clause
rel(X,N,and(N,R)):- {that}, antecedent(X) -: sent(R).

% having found a relative pronoun; find a sentence assuming
% X as the antecedent of a missing np

vp(X,P) :- intrans_vb(X,P).
vp(X,P):- trans_vb(X,Y,P1), np(Y,P1,P).
vp(X,P):- bitrans_vb(X,Y,Z,P1), np(Y,P1,P2), np(Z,P2,P).

intrans_vb(X,won(X)):- {won}.
trans_vb(X,Y,saw(X,Y)):- {saw}.
bitrans_vb(X,Y,Z,gave(X,Y,Z)):- {gave}.

{W}:-dcg_connect(W). % syntactic sugar for dcg_connect
```

The difference with usual DCGs is that only *terminals* act as *state transformers*. *Nonterminals* carry no existential variables and are implemented by ordinary predicates. Output-construction, carried out in this example explicitly, can be left as well to a second DCG-stream, while a third stream can be used as a stack accumulating information for future use e.g. in anaphora resolution. For debugging purposes as well as for meta-interpretation in general, the absence of explicit (single or multiple) DCG arguments is an advantage.

5 Resolving anaphoric reference in discourse

One of the most difficult problems in language processing is that of determining which entities co-specify. For instance, in "John photographed Mary, and then she photographed him", we can think of the proper names "John" and "Mary" as specifying entities in the world, and of the pronouns "him" and "she" as co-specifying the same entities. The notion of co-specification, first introduced in [Sid81] avoids the pitfalls of previous notions of antecedence, since a pronoun might follow rather than precede its "antecedent" (as in "When he sleeps, John snores"), or even be inexplicit (as in "I was caught speeding, but he only gave me a warning"). Notice also that co-specification may be ambiguous (as in "John photographed Tim, and then he photographed Mary") and may not respect syntactic agreement (as in "If a student is admitted, they must satisfy the acceptance criteria").

It is interesting that the extensions to logic grammars that we present in this paper admit two different methodologies for resolving co-specification: the use of multiple accumulators, and the use of linear and intuitionistic implication. The first one was investigated in [FDT95], so we will only briefly summarize it here. Then we shall introduce the hypothetical reasoning approach.

5.1 Co-specification Resolution through Multiple Accumulators

As a discourse is processed, specifiers can be marked for later consultation as potential co-specifiers through the use of accumulators. To make co-specifier resolution as general as possible, all the information pertaining to each specifier encountered is accumulated in a stack. During the resolution process, this stack contains a list of *contextual* referents, ordered temporally (i.e. potential co-specifiers derivable from the context/discourse). As an example, after processing the first sentence of the discourse: "John greeted Fred. He then greeted Cathy.", the stack contains specifiers for John and Fred. When the pronoun *he* is being resolved, it may potentially co-specify with either of these entities. Constraints on matching can be used to select both possibilities, ordered according to preference criteria (so that *John* may be selected before *Fred*).

When processing a discourse, we may need to check if a definite specifier refers to one of the entities in the stack. This searching process can be directed by providing a number of *match skeletons* that specify syntactic, semantic and contextual constraints that co-specification (i) *must* agree with (required), (ii) would *preferably* agree with (prefered) and (iii) *must not* agree with (forbidden). For the pronoun *she*, any co-specifier must be feminine and singular, whereas for *this dog*, the semantic type of any referent must be compatible with *dog*. A theory of anaphoric co-specifiers may dictate that pronouns should try to match on focus [Sid81] or case. Since there may be multiple sets of preferred constraints, the matching process can be given a list of preferred skeletons, ordered by preference.

The co-specifier matcher combines each preferred skeleton with the required skeleton, and then searches the stack for the first matching specifier which does not also match the forbidden skeleton. We also search for a potential match using only the required information. Each match is added to the list of potential co-specifiers, ordered using the same order given by the preferred constraints.

By determining the entire list of potential co-specifiers in one step, we avoid returning the same specifier twice. Also, it may not be possible to express all the contextual and semantic constraints in the skeletal information given to the matching process, leading to two problems: (i) the list returned may not be ordered properly and (ii) it may contain superfluous co-specifiers. Given this preliminary list, higher-level processes may then weed out impossible co-specifiers and impose additional ordering constraints to select from the remaining entities (i.e. to choose a preferred reading in the face of ambiguity). A similar scheme is proposed in [Hob86], in which a lower level process generates a set of candidate referents for pronouns, and a higher level process selects one antecedent from this set using relative criteria.

To illustrate this process, suppose we use a theory of anaphora resolution that requires matching on gender and number, and prefers to match on case. For the above discourse, we would return the list [*John, Fred*], since both satisfy the required gender and number constraints, but John matches on the nominative case. This simple scheme fails, however, for the discourse: "Cathy was talking with Fred. When Fred saw Sharon, she ignored him." When resolving *she*, the list returned is [*Cathy, Sharon*]. If we prefer to match on Sharon because she is mentioned later than Cathy, we can process the returned list, reversing the preference order based on the temporal ordering at the sentential or topic level. Expressing general constraints such as temporal or topical proximity is better done on the resulting list rather than in the matching process, particularly if such constraints involve relative comparisons among potential co-specifiers.

If a system also provides a set of semantic constraints (for e.g. on the possible object and subject of a verb), these can be used to prune or order the potential co-specifiers. To illustrate, consider the sentences: “When John saw the dog, he was barking.” and “When John saw the dog, he was laughing.” If our semantic constraints restrict the agent of barking to be a dog and the agent of laughing to be a person, then resolving *he* in both sentences unambiguously selects the correct co-specifier. If we permit people to bark and dogs to laugh as unlikely, but possible scenarios, then the co-specification resolver will return both *John* and *dog* as potential co-specifiers for *he*. In the first case, the preferential order will be [*dog, John*], whereas in the second case it will be [*John, dog*]. Many of these semantic constraints can be made available, for example, in the virtual hierarchy.

The above scheme can automate many forms of co-specification. In [FDT95], we describe several extensions which improve the flexibility and generality of the process. These extensions permit the specification of additional constraints to control the matching process, and to derive new specifiers from mentioned entities (e.g. to *generalize* a class from an instance, as in “John is a marathon runner. They have a lot of endurance.” or to *specialize* an instance from a class as in “Harlequin ducks are beautiful. I saw one yesterday.”).

5.2 Hypothetical reasoning vs. Hidden Accumulator Grammars

An alternative way of storing the information gleaned from the grammar and the noun phrases as they appear is through temporarily adding this information as hypotheses ranging over the current continuation. Consulting it then reduces to calling the predicate in which this information is stored.

To exemplify, let us consider a very simple grammar which retrieves the list of all head nouns as the semantic representation of its sentences. Consider the discourse:

```
John walks.
Amy gives a computer to him.
```

Upon encountering/generating the noun phrase “John”, we can store the information found in the lexical rule for “John” with a linear implication.

When we then encounter, say, a pronoun, we can compare its syntactic features with those of the candidate co-specifiers (those stored in “candidate” hypotheses). We refer to Appendix II for the actual implementation.

Notice that by scoping the implications over the current continuation, the noun phrases of all previous sentences in the discourse are available for resolving co-specification. Once a potential referent is found, it is stored as a feature of the pronoun in question. Thus each pronoun will, at the end of the analysis, carry the whole list of its potential referents as a feature. User-defined criteria can then be used, as in the multiple accumulator approach, to further refine the list of candidate co-specifiers.

We have defined the co-specifier predicate so that the representations of the noun phrases are the only candidates allowed. It is interesting to point out that in order to handle abstract co-specifiers [Ash93], such as events or propositions, all we have to do is extend the definition so that other parts of a sentence can be identified as possible specifiers as well.

The extensive use of taxonomic information motivates the need for a principled approach, similar in objectives with the sorts in Life, but handled through explicit operations, as an abstract data type. This is the subject of the next section.

6 Virtual Type Hierarchies

Hierarchical, or taxonomic, information is pervasive in many areas of inquiry. In natural language processing, taxonomies have been used for semantic type hierarchies, in which selectional constraints can be imposed, and for word class hierarchies, in which syntactic constraints can be specified [FDT95, PS87]. An analysis of the use of taxonomies in unification-based grammars which exploit typed feature structures is given in [Car92].

The main operations on taxonomies are *greatest lower bound*, *least upper bound* and *subsumption checking*, in addition to the specification of inheritable properties. For in-depth coverage of how large taxonomies can be managed in order to make these operations efficient, see [AKBLN89, Fal94].

In the present work, we build on research in [FDT95] to extend the application of hierarchical information in natural language processing. As an example, consider the discourse *"The dog saw the mailman. He was barking."*. We can construct a hierarchy of semantic types, in which mailmen are a subset of people, and the inheritable property that the agent of *barking* is usually a dog. This information would effectively rule out *the mailman* as referring to the same individual as *he* unless there was additional contextual or world knowledge to believe that the mailman might bark.

6.1 Sparse Logical Terms

There are two uses of variables in Herbrand terms: as place holders and for co-reference. Since Herbrand terms have fixed arity, the only purpose of anonymous variables is to fill unspecified positions. Sparse terms [Fal95] by default have variable arity and so remove the need for anonymous variables. The implicit position information of Herbrand terms is lost, and so must be explicitly stated. A similar situation arises in sparse matrix representation, on which sparse terms are modeled. As an example, the term: a(b(_,c,_,_,X),_,_,d(X,_,e(_,_),_)) is represented as a sparse term by: `a~[1-b~[2-c,5-X],4-d~[1-X,3-e]]`. To be more formal, we provide a concrete definition:

Definition 6.1 *A* sparse term *is either (i) an atom (ii) a named variable or (iii) a functor of the form* `a~L`*, where* `a` *is the functor symbol and* `L` *is a sparse argument list. A sparse argument list is a list of elements of the form* `n-ST`*, where* `ST` *is a sparse term and n is the index of* `ST` *in the parent term. This list is ordered by increasing indices with no repetitions.*

One of the advantages of sparse terms is the ease with which they may be extended. We describe below some these extensions that are useful for natural language processing.

Binding Arity: Since arity is variable in the above representation, there is not a one-to-one correspondence between sparse and ordinary terms. For example, the following terms correspond to the sparse term `f~[1-a]`: f(a), f(a,_), f(a,_,_), f(a(_),_), ... We can explicitly bind the arity of a term by extending part (iii) of our definition to allow functors of the form `a/N~L` where `N` is the arity of the functor. As an example, the term f(_,b(_,_),c,d(e,_),_) would be completely represented by `f/5~[2-b/2,3-c/0,4-d/2~[1-e/0]]`.

Anonymous Functors: An interesting variation allows terms to specify only those argument positions which are occupied, but not record the functor or atom in that position. This is useful to maintain uncertainty or to avoid repeating functor information for terms that will later be unified. To provide functorless terms, we simply remove the functor or atom from the elements of the sparse argument list. The term f(_,b(_,_),_,c(d,_,e),_) would thus be represented as the term `[2,4-[1,3]]`.

Attribute Indices: The indexing in Prolog terms is implicitly numeric. Since sparse terms maintain indexing explicitly, a trivial extension allows atomic rather than just numeric indices. This provides some of the functionality of ψ-terms [AKP93] and permits implementation of attribute-value matrices, or feature structures, as used in several computational linguistic systems (e.g. [Car92, PS87]). A predicate can be provided to access the value of an attribute, or a sequence of attributes.

Disjunctive Functors: Thus far, we have permitted two levels of certainty regarding a functor symbol: either it is unknown (i.e. it may be any atomic symbol) or it is known. Between these extremes lies a range of increasingly focused information as to the actual functor symbol. That is, we may know that it is one of a set of possible symbols. When this set has cardinality one, we know which symbol it must be. We will name such functors *disjunctive* and represent them with a set notation. For example, the term `[model-{MacSE; MacII}, memory-{1;2;4;8}]` may be used to represent a computer system whose model type is either a MacSE or a MacII and with either 1, 2, 4 or 8 KB of memory.

Disjunctive Terms: For similar reasons, we may wish to maintain uncertainty at the term level (either for entire terms, or for subterms within a term). This is also accomplished using the set notation. As an example, the term `bark~{[person-{first;second}]; [number-plural,person-third]}` may be used to represent the word *bark* as a first person, second person, or third person plural verb. This can be later constrained to, for e.g., third person, as in:

```
     | ?- sortDef(bark,S,_), unifyST([person-third],S,T).

S = {bark~[person-{first;second},sem-s_bark~[agent-s_dog]];
     bark~[person-third,number-plural,sem-s_bark~[agent-s_dog]]},
T = bark~[person-third,number-plural,sem-s_bark~[agent-s_dog]] ?
```

Generalized Coreference: For coreference, LIFE uses more generalized coreference labels than the named variables of Prolog. These labels can specify coreference between any two locations in the graphical representation, not just between leaves.

In order to synthesize these enhancements, we give the following definition of sparse terms:

SparseTerm	:=	CorefLabel:ST | CorefLabel: | ST
		| CorefLabel:{ST; ...;ST} | {ST; ...;ST} (where cardinality is > 1)
CorefLabel	:=	String
ST	:=	Functor | Functor~ArgList | ArgList
Functor	:=	Atom | Atom/Arity | /Arity | {Atom; ...;Atom}
		| {Atom, ..., Atom}/Arity (where cardinality is > 1)

ArgList	:=	[] | [Argument | ArgList]
Argument	:=	Index-SparseTerm
Index	:=	String (i.e. an attribute indicating the argument position).

In order to use sparse terms, the predicate **`unifyST(T1, T2, T)`** unifies two input terms and produces the unified term T. We also provide predicates for antiunification and subsumption checking.

Our representation shares some commonality with the ψ-terms in LIFE [AKP93], in particular attribute indexing, unbound arity and coreference labels, but it also differs in several respects. In particular, sparse terms deviate from ψ-terms for binding arity, anonymous functors and disjunction. Another significant difference is that our representation is intended as an enhancement to Prolog systems, not as a replacement.

6.2 Sparse Term Taxonomies

We have implemented predicates that permit the construction of hierarchies, in which each node of the hierarchy has associated with it a sort and a sparse term that encodes the default properties for entities of that type. These predicates provide much of the functionality of LIFE constrained sorts [AKP93]. The top node in the hierarchy is *top* (i.e. $\top$), and may not be modified.

The predicate **`subsort(Subsort, Supersort)`** adds information that *Subsort* is a subsort of *Supersort*, provided that the partial order structure of the hierarchy is not violated. This is equivalent to the "`<|`" predicate in LIFE. The default properties of the supersort are unified with those of the subsort (if it had any prior to calling the predicate) using what we call *c-unification*. In c-unification, one of the terms is *dominant* and the other is *subordinate*. During unification, if a conflict arises (in which ordinary unification would fail), only the information in the dominant term is kept. When specifying subtyping, the pre-existing properties of a sort dominate those of the new supersort. In this way, non-monotonic property inheritance is achieved.

Properties are associated with sorts using the **`set_property(Sort,Properties)`** predicate. This is similar to the "`::`" predicate in LIFE. Properties are specified as sparse terms, and are c-unified with the existing properties of the sort. New properties are assumed to dominate previously defined properties.

There are several ways to access property terms associated with sorts. Sorts are stored in predicates of the form **`sortDef(Sort,Property,Parents)`**, which may be directly accessed. In order to increase time efficiency, the Property term will contain all inherited properties, but to save on space and to allow for recursive properties, sorts within properties are not expanded. A fully expanded property term can be obtained using the predicate **`expandSort(Sort,FullProperty)`**, but this will not terminate on cyclic properties.

To handle recursive properties, we implemented *lazy expansion* of sort properties. LIFE also does a lazy expansion of terms, but in a slightly different way than here. As soon as an attribute of a subterm which is an unexpanded sort is accessed, the subterm is expanded using the sort's property term. To illustrate, consider the following definitions, the first of which states that every person has a mother and birthdate:

```
subsort(person, top).
set_property(person, [mother-person, birthdate]).
```

```
subsort(joe, person).
set_property(joe, [mother-anne~[mother-betty]]).
```

This builds a simple hierarchy of people, in which all people have a mother who is a person. Clearly, fully expanding the person sort is not possible. Instead, the property terms of "person" and "joe" are:

```
| ?- sortDef(person,S,T).

S = person~[mother-person,birthdate],
T = [top] ?

| ?- sortDef(joe,S,T).

S = joe~[mother-anne~[mother-betty~[mother-person,birthdate],birthdate],
         birthdate],
T = [person] ?
```

As can be seen, only a minimal amount of expansion is performed. We must expand the leaf "betty" since the only connection between "betty" and the sort "person" is through the property specified for "joe". Since "betty" is not a sort, if we don't expand during the property specification, we lose this connection.

Predicates are also provided to access and set values along particular chains of attributes. For example, suppose we wish to get the "mother of the mother" attribute of some property term. This is done through a projection predicate `projectST(AttributePath, Term, Value)`, where the attribute path is specified by a list of attributes:

```
| ?- sortDef(joe,S,_), projectST([mother, mother], S, X).

S = joe~[mother-anne~[mother-betty~[mother-person,birthdate],birthdate],
         birthdate],
X = betty~[mother-person,birthdate] ?
```

Setting the value at the end of an attribute path is done analogously to unification, using the predicate `setProjectST(AttributePath,Value,TermIn,TermOut)`, as in the following examples:

```
| ?- sortDef(joe,S,_), setProjectST([mother,mother], jane, S,X).

S = joe~[mother-anne~[mother-betty~[mother-person,birthdate],birthdate],
         birthdate],
X = joe~[mother-anne~[mother-jane~[mother-person,birthdate],birthdate],
         birthdate] ?
```

The use of these predicates for the construction and use of a sort hierarchy can be seen in Appendix II.

7 Related work

Note that existing work on Linear Logic based Natural Languages processing [Hod92a, Hod92b, HM92, PM90] is mostly at sentence level, while, ours covers text level constructs. This is made easy by using hypothetical assumptions which range over the current continuation, instead of locally scoped implications.

There is some commonality between our approach to co-specifier resolution and the pronoun anaphora approach in the public domain LIFE natural language analyzer. In particular, both schemes employ a hierarchy of semantic types in order to impose selectional constraints (although the constraints themselves are specified using functions in the the LIFE program, instead of as inheritable properties as in our system), and both use accumulators to manage potential co-specifiers (although these are hand-coded in the LIFE program). There are, however, several key differences in the approaches. First, anaphora resolution in the LIFE program is based on antecedence, while our approach uses the more general and less problematic notion of co-specification [Sid81]. Second, the resolution process in the LIFE program is fixed within the grammar rules: pronoun resolution simply searches the temporally ordered list of potential co-specifiers for the first match on gender, number and semantic type. Our approach has been designed for flexibility: although some matching constraints may be specified in the grammar rules, most are specified lexically. This allows a range of matching constraints and also permits matching on abstract entities, as in *"John kicked Fred on Monday, and it hurt"* [Ash93]. Finally, we have added a number of extensions to the matching process which allow the determination of implicit referents, as in *generalization* and *specialization*.

8 Future work

The 'world-modeling' program is expected to grow to a realistic MUD-server using BinProlog's distributed programming facilities (Linda based blackboard). An obvious application of our techniques is to various planning problems. It would be also interesting to explore the interaction between our hypothetical reasoning tools and constraint solvers. In practice this means porting them to a finite domain constraint solver like WAMCC [CD95], which also features global variables and backtrackable destructive assignment.

Our tools are in an experimental stage, a tighter integration will follow. The mapping from sparse terms to BinProlog's logical global attributed variables will allow 'compiling' them to WAM-code and ultimately to C, therefore removing the remaining inefficiencies in some interpreted operations. We also intend to continue exploring the uses of sparse terms and taxonomies in natural language processing.

9 Conclusion

We have a presented a framework which uses various hypothetical reasoning techniques implemented as backtrackable state information in some typical natural language processing problems. Their synergy is needed for reasons of expressiveness and flexibility.

The novelties are:

- linear implications scoped over the current continuation (i.e. the remaining AND-branch of the resolution) allow extension beyond individual sentences

- although they can be seen as particular instances of linear assumptions [TDF95] Hidden Accumulator Grammars are an efficient and flexible tool, allowing enforcement of chronological order on linear assumption (if needed) and a successful replacement for DCGs, both on efficiency and ease of use grounds

- HAGs and lazy expansion of taxonomies gives to a generic Prolog engine some of the expressiveness of Life extensions (accumulators and sorts)

Compared to previous frameworks based on Linear (Intuitionistic) Logic, ours is portable and runs on top of generic Prolog systems. This is a practical advantage over systems like Lolli or λProlog. Backtrackable destructive assignment, when encapsulated in higher-level constructs turns out to simplify and possibly replace widespread idioms like DCGs while offering more powerful facilities in the form of hypothetical assumptions and multiple accumulators. This also reduces the need for explicitly imperative constructs like *assert* and *retract* in logic programming languages.

References

[AD89] H. Abramson and V. Dahl. *Logic Grammars*. Symbolic Computation AI Series. Springer-Verlag, 1989.

[AKBLN89] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr. Efficient implementation of lattice operations. *ACM Transactions on Programming Languages*, 11(1):115–146, 1989.

[AKP93] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16(3/4):195, 1993.

[Ash93] N. Asher. *Reference to Abstract Objects in Discourse*, volume 50 of *Studies in Linguistics and Philosophy*. Kluwer, 1993.

[Car92] B. Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, London, England, 1992.

[CD95] Philippe Codognet and Daniel Diaz. Compiling Prolog to C : the WAMCC system. In *Proceedings of the 12-th International Conference on Logic Programming*, Yokohama, Japan, 1995. The MIT Press. to appear in ICLP95.

[Dah91] V. Dahl. Incomplete types for logic databases. *Applied Mathematical Letters.*, 4(3):25–28, 1991.

[Fal94] A. Fall. The foundations of taxonomic encoding. *Submitted to ACM Transactions on Programming Languages. Also available as Simon Fraser Universtity Technical Report SFU LCCR TR 94-20*, 1994.

[Fal95] A. Fall. Sparse logical terms. *To appear in Applied Mathematical Letters.*, 1995.

[FDT95] A. Fall, V. Dahl, and P. Tarau. Resolving co-specification in contexts. In *Proc. Workshop on Context in Natural Language Processing (to appear)*, Montreal, Canada, 1995.

[HM92] J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 1992.

[Hob86] J. Hobbs. Resolving pronoun references. In *Readings in Natural Language Processing*, pages 339–352. Morgan Kaufmann Publishers, Inc., 1986.

[Hod92a] J. Hodas. Specifing Filler-Gap Dependency Parsers in a Linear-Logic Programming Language. pages 622–636, Cambridge, Massachusetts London, England, 1992. MIT Press.

[Hod92b] J Hodas. Specifying Filler-Gap Dependency Parsers in a Linear Logic Programming Language. In Krzysztof Apt, editor, *Proc. 1992 Joint International Conference and Symposium on Logic Programming*, pages 622–636. MIT Press, 1992.

[PM90] R. Pareschi and D.A. Miller. Extending definite clause grammars with scoping constructs. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, pages 373–389, Jerusalem, Israel, 1990. MIT Press.

[PS87] C. Pollard and I. Sag. *Information-Based Syntax and Semantics.* CSLI Lecture Notes No. 13. Stanford, CA, 1987.

[Sid81] C. Sidner. Focussing for Interpretation of Pronouns. *American Journal for Computational Linguistics*, 7(4):217–231, 1981.

[Tar95] Paul Tarau. BinProlog 3.30 User Guide. Technical Report 95-1, Département d'Informatique, Université de Moncton, February 1995. Available by ftp from *clement.info.umoncton.ca.*

[TDF95] Paul Tarau, Veronica Dahl, and Andrew Fall. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. Technical Report 95-2, Département d'Informatique, Université de Moncton, April 1995. Available by ftp from *clement.info.umoncton.ca.*

[Van89] Peter Van Roy. A useful extension to Prolog's Definite Clause Grammar notation. *SIGPLAN notices*, 24(11):132–134, November 1989.

Appendix I - World modeling with ‘executable English’

```
:-[library(tree)].

:-op(50,fy,green).
:-op(50,fy,red).
:-op(50,fy,yellow).
:-op(50,fy,blue).

:-op(100,fy,a).
:-op(100,fy,an).
:-op(100,fy,the).
:-op(100,fy,his).
:-op(100,fy,her).
:-op(200,xfy,and).
:-op(300,xfx,to).
:-op(300,fx,to).
:-op(300,xfx,in).
:-op(300,fx,in).
:-op(700,xfx,is).
:-op(700,xfx,gives).
:-op(700,xfx,takes).
:-op(700,xfx,drops).
:-op(700,xfx,goes).
:-op(700,xfx,has).
:-op(970,fx,if).
:-op(980,xfy,then).
:-op(990,xfy,else).

check(X):-assumed(X),assumel(X).

% syntax for free, with operator precedence parsing...

text(there is a book in the green room).
text(mary has a flower).
text(joe is in the green room).
text(he has a diamond).
text(she is in the blue room).
text(he takes the book).
text(he goes to the blue room).
text(he gives it to her).
text(mary drops the book).
text(he takes it).
text(he goes to the red room).
text(she goes to the red room).
text(she gives a flower to him).
text(
  if joe has the moon
  then joe gives the moon to mary
  else joe gives his diamond to her
).
text(joe goes to the green room).
```

```
show:-text(X),write(X),nl,display(X),nl,nl,ppt(X),nl,nl,fail.

% world-state (context) modeling

go:-
   findall_conj(action(T),text(T),Ts),
   Ts.

action(A):-
   write('READING: '),write(A),nl,
   ppt(A),nl,
   get_action(A,Pred),
   (Pred->write('SUCCEEDS: '),write(Pred);ppt('FAILING!!!'(Pred))),
   nl,nl,
   write('ABSTRACT ASSUMPTIONS:'),nl,listing,
   nl,write('--------------------------------------'),nl,nl.


get_action(A is B,Action):-!,get_action1(action_is(A,B),Action).
get_action(A has B,Action):-!,get_action1(action_has(A,B),Action).
get_action(Term,Action):-get_action1(Term,Action).

get_action1(Term,Action):-
   Term=..[F|Xs],
   abs_list(Xs,Ys),
   Action=..[F|Ys].

get_constraint(Term,Cond):-get_action1(Term,Cond).

action_is(there,O0 in P0):-!,abs1(O0,O),abs1(P0,P),assumel(P has O).
action_is(S0,O):-p_nom(S0,S),assumel(O has S).

action_has(S0,O0):-p_nom(S0,S),p_acc(O0,O),assumel(S has O).

takes(S0,O0):-
  p_nom(S0,S),p_acc(O0,O),
  check(P has S),
  transfer(O,P,S).

drops(S0,O0):-
  p_nom(S0,S),p_acc(O0,O),
  check(P has S),
  transfer(O,S,P).

gives(S0,O0 to C0):-
  p_nom(S0,S),p_acc(O0,O),p_acc(C0,C),
  transfer(O,S,C).

goes(S0,O0):-
  p_nom(S0,S),p_acc(O0,O),
  transfer(S,_,O).

% this has ordinary Prolog (closed world) semantics
% handling counterfactuals needs abductive reasoning
else(then(if(If0),Then0),Else0):-
```

```
  get_constraint(If0,If),get_action(Then0,Then),get_action(Else0,Else),
  (If->Then;Else).

then(if(If0),Then0):-
  get_constraint(If0,If),get_action(Then0,Then),
  (If->Then).

transfer(O,From,To):-
   assumed(From has O),
   assumel(To has O).

abs_list([],[]).
abs_list([X|Xs],[Y|Ys]):-abs(X,Y),abs_list(Xs,Ys).

abs(X to Y,AX to AY):-!,abs1(X,AX),abs1(Y,AY).
abs(X in Y,AX in AY):-!,abs1(X,AX),abs1(Y,AY).
abs(X,AX):-abs1(X,AX).

abs1(X,Z):-compound(X),abs0(X,Y),!,abs(Y,Z).
abs1(X,X).

% abstraction operator: reduces to semantic roots
abs0(a(X),X).
abs0(an(X),X).
abs0(the(X),X).
abs0(in(X),X).
abs0(his(X),X).
abs0(her(X),X).
abs0(to(X),X).

p_nom(P,X):-find_if(p_nom0(P,Xs),member(X,Xs),P=X).

p_nom0(he,[joe]).
p_nom0(she,[mary]).
p_nom0(it,[book,flower]).

p_acc(P,X):-find_if(p_acc0(P,Xs),member(X,Xs),X=P).

p_acc0(him,Xs):-p_nom0(he,Xs).
p_acc0(her,Xs):-p_nom0(she,Xs).
p_acc0(it,Xs):-p_nom0(it,Xs).

find_if(If,Then,Else):-
  findall(If-Then,If,IfThens),
  ( IfThens=[]->Else
  ; member(If-Then,IfThens),
    Then
  ).
```

Appendix II. A natural language analyzer/generator

```
/***************************************************************************
Possible 'stories' in generation mode:

?- go.

[john,walks,.,a,woman,laughs,.,a,woman,that,laughs,gives,a,computer,to,him]
...
[john,walks,.,a,man,buys,a,book,.,a,man,that,buys,a,book,gives,it,to,him]
***************************************************************************/

:-dynamic pred/1,gap/1.

go:- top(T),member(that,T),member(him,T),write(T),nl,fail.

top(Xs):-
    dcg_def(Xs),
    interesting-:interesting-:interesting-:text,
    dcg_val([]).


{W}:-dcg_connect(W). % syntactic sugar

+(W):-assumel(W). % syntactic sugar
*(W):-assumei(W).

% grammar

text:-interesting,sent,rest.

rest.
rest:-{.},text.

% the number of assumed pred/1 limits the amount of anaphoric references

sent:-sent(Args), + pred(Args), + pred(Args).

sent([S,V]):-
  s(S),  should_be(S,kind,K),
         should_be(S,case,subject),
  v(1,V),should_be(V,agent,K).
sent([S,V,O]):-
  s(S),  should_be(S,kind,KA),
         should_be(S,case,subject),
  v(2,V),should_be(V,agent,KA),
         should_be(V,object,KO),
  o(O),  should_be(O,kind,KO),
         should_be(O,case,object).
sent([S,V,O,C]):-
  s(S),  should_be(S,kind,KS),
         should_be(S,case,subject),
  v(3,V),should_be(V,agent,KS),
         should_be(V,object,KO),
         should_be(V,co_agent,KC),
  o(O),  should_be(O,kind,KO),
         should_be(O,case,object),
  {to},
  c(C),  should_be(C,kind,KC),
            should_be(C,case,object),
  S\==C.

s(X):-n(X).
o(X):-n(X).
c(X):-n(X).

v(N,X):-{X},should_be(X,args,N).

n(X):-gap([X|_]),!.
n(X):-pn(X).
n(X):-in(X).
n(X):-cn(X).
n(X):-cn(X),rel([X|_]).

pn(P):-{P},anaphora(P).
in(X):-{X},should_be(X,class,individual).
cn(X):-{a},{X},should_be(X,class,concept).
cn(X):-{the},{X},should_be(X,class,concept),known(X).

anaphora(P):-
  known(X),
  should_be(X,gender,G),
  should_be(P,gender,G),
  should_be(P,class,pronoun).

known(S):-pred([S|_]).
known(O):-pred([_,_,O|_]).

rel(Args):-
  rp,
  pred(Args),
  gap(Args) -: sent(Args).

rp:-{that}.

should_be(Word,Attribute,Value) :- projectST([Attribute],Word,Value).

% noun sorts

:- subsort(thing, top).
:- set_property(thing, [class-concept, gender-neutral, kind-thing]).
:- subsort(name, top).
:- set_property(name, [class-individual]).
:- subsort(animate, thing).
:- set_property(animate, [kind-animate]).
:- subsort(inanimate, thing).
:- set_property(inanimate, [kind-inanimate]).

:- subsort(person, animate).
:- subsort(woman, person).
:- set_property(woman, [gender-female]).
:- subsort(man, person).
:- set_property(man, [gender-male]).
:- subsort(john, name).
:- subsort(john, man).
:- subsort(mary, name).
:- subsort(mary, woman).
:- subsort(book, inanimate).
:- subsort(compute, inanimate).


% pronouns

:- subsort(he, animate).
:- set_property(he, [case-subject, class-pronoun, gender-male]).
:- subsort(him, animate).
:- set_property(him, [case-object, class-pronoun, gender-male]).
:- subsort(she, animate).
:- set_property(she, [case-subject, class-pronoun, gender-female]).
:- subsort(her, animate).
:- set_property(her, [case-object, class-pronoun, gender-female]).
:- subsort(it, thing).
:- set_property(it, [class-pronoun]).


% verb sorts

:- subsort(verb, top).
:- subsort(intransitive_verb, verb).
:- set_property(intransitive_verb, [args-1]).
:- subsort(transitive_verb, verb).
:- set_property(transitive_verb, [args-2]).
:- subsort(bitransitive_verb, verb).
:- set_property(bitransitive_verb, [args-3]).

:- subsort(walks, intransitive_verb).
:- set_property(walks, [agent-animate]).
:- subsort(laughs, intransitive_verb).
:- set_property(laughs, [agent-animate]).
:- subsort(buys, transitive_verb).
:- set_property(buys, [agent-animate, object-thing]).
:- subsort(gives, bitransitive_verb).
:- set_property(gives, [agent-animate, co-agent-animate, object-thing]).
```