Assumptive Logic Programming

Veronica Dahl
Logic and Functional Programming Group
School of Computing Science
Simon Fraser University
Burnaby, B.C. Canada V5A 1S6
veronica@cs.sfu.ca

Paul Tarau
University of North Texas
Dept. of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

Abstract

We introduce Assumptive Logic Programming (**ALP**), a novel framework of handling state information for logic programming languages on top of backtrackable assumptions (linear affine and intuitionistic implications ranging over the current continuation) and we show its applications to hypothetical reasoning and high level natural language processing.

In particular, we show how to emulate abductive logic programming in terms of assumptions, we specify Assumption Grammars (**AGs**) in terms of ALP primitives and show an example of their use for natural language processing (anaphora resolution). Finally we provide a (surprisingly simple) implementation of link grammars in terms of Assumption Grammars.

Keywords: *logic programming extensions, hypothetical reasoning, state in logic programming, logic grammars, natural language processing, linear and intuitionistic assumptions.*

1 Introduction

In human communication, assumptions play a central role. Linguists and logicians have uncovered their many facets. For instance, the assumption of a looking glass' existence and unicity in "The looking glass is turning into a mist now" is called a *presupposition*; the assumption that a polite request, rather than a literal question, will be "heard" in "Can you open the door?" is an *implicature*.

Much of AI work is also concerned with the study of assumptions in one form or another: abductive reasoning "infers" (assumes) p from q given that the reverse implication `p => q` holds; uncertain information with high probability is used (assumed) in some frameworks, and so on.

Independently, work on intuitionistic logic and linear logic [4] has provided formally characterized embodiments of assumptions which have been influential on logic programming and logic grammars [9, 10, 6]. The result is not only a better understanding of their proof-theoretical characteristics but also a growing awareness on the practical benefits of integrating them in conventional Prolog systems.

Different types of logic grammars have evolved through the years, motivated in such concerns as ease of implementation, further expressive power, a view towards a general treatment of some language processing problems, such as anaphora resolution, or towards automating some part of the grammar writing process, such as the automatic construction of parse trees and internal representations. Generality and expressive power seem to have been the main concerns underlying all these efforts.

As the apparently simple translation scheme of grammars to Prolog became popular, DCGs have been assimilated by means of their preprocessor based implementation. When restricted to definite clauses the original DCG translation is indeed operationally trouble free and has a simple Herbrand semantics. On the other hand, mixing DCGs with full Prolog and side effects has been a prototypical Pandora's box, ever since. Cumbersome debugging in the presence of large list arguments of translated DCGs was another initially unobvious consequence, overcome in part with depth-limited term printing. The complexity of a well-implemented preprocessor made almost each implementation slightly different from all others. The lack of support for "multiple DCG streams", although elegantly solved with Peter Van Roy's Extended DCGs [13], required an even more complex preprocessor and extending the language with new declarations. Worse, proliferation of programs mixing DCG translation with direct manipulation of grammar arguments have worked against data abstraction and portability.

Translation free, and more general than Van Roy's Extended DCGs [13], assumption grammars consist of logic programs augmented with a) multiple implicit accumulators, useful in particular to make the input and output strings invisible, and b) linear and intuitionistic implications scoped over the current continuation (i.e., over the remaining AND branch of the resolution), based on a variant of *linear logic* [4] with good computational properties, *affine* logic [8].

Handling state information in Assumption Grammars is a special case of the general problem of cleanly and efficiently handling state information in declarative languages. We shall therefore examine this next, as well as the related problem of supporting backtrackable state changes.

2 State and scope in logic languages

The main problem is that expressing change contradicts some of the basic principles logic (and functional) languages are built on.

In Prolog, the *implicit arguments* of DCG grammars correspond to a chain of variables, having exactly two occurrences each, as in `a(X1,X4) :- b(X1,X2), c(X2,X3), d(X3,X4)`. We can see the chain of variables as successive states of a unique object. In the presence of *backtracking*, previous values must be kept for use by *alternative* branches. Although irrelevant to the user, for the implementor, this situation conflicts with possibility of *reuse* and makes single-threaded objects more complex in non-deterministic LP languages than in committed choice or functional languages.

In functional languages like Haskell where, in a deterministic framework, elegant unified solutions have been described in terms of monads and continuations, 'imperative functional programming' is used (with relative impunity) for arrays, I/O processing, etc. For non-deterministic logic programming languages like Prolog, the natural *scope* of declarative *state* information is the current AND-continuation as we want to take advantage of re-usability on a deterministic AND-branch in the resulting tree-oriented resolution process.

This suggests that we need the ability of extending the scope of a state transition over the current *continuation*, instead of keeping it local to the body of a clause. To achieve this our *linear* and *intuitionistic* assumptions will be *scoped* over the *current continuation.*

Implementing scope Once a backtrackable *assume* primitive is implemented with assumptions ranging over the the current AND-branch, it is easy to make unavailable a given assumption to an arbitrary future segment in the current continuation by binding a logical variable serving as a *guard.*

3 Hypothetical reasoning with linear and intuitionistic assumptions

This framework will cover a fairly general form of backtrackable state information, which increases the expressiveness of a Prolog system while reducing visual noise due to useless argument passing. Our proposed Assumption Grammars will be derived as an instance of the framework.

3.1 Assumed code, intuitionistic and linear affine implication

We will give a short description of the primitive operations and point out some of the differences with other linear/intuitionistic logic inspired implementations.

Intuitionistic `assumei/1` adds temporarily a clause usable in subsequent proofs. Such a clause can be used an indefinite number of times, like asserted clauses, except that it vanishes on backtracking. The assumed clause is represented on the heap.

Its scoped versions `Clause=>Goal` and `[File]=>Goal` make `Clause` or respectively the set of clauses found in `File`, available only during the proof of `Goal`. Clauses assumed with `=>` are usable an indefinite number of times in the proof, e.g. `a(13) => (a(X),a(Y))` will succeed.

Linear assumel/1 adds a clause usable *at most once* in subsequent proofs. Being usable *at most once* distinguishes *affine* linear logic from Girard's original framework where linear assumptions should be used *exactly* once. This assumption also 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. They vanish on backtracking and each clause is usable at most once in the proof, i.e. `a(13) -: (a(X),a(Y))` will fail. Note however, that `a(13) -: a(12) -: a(X)` will succeed with `X=12` and `X=13` as alternative answers, while its non-affine counterpart `a(13) -o a(12) -o a(X)` as implemented in Lolli or Lygon, would fail.

We can see the `assumel/1` and `assumei/1` builtins as linear affine and respectively intuitionistic implication scoped over the current AND-continuation, i.e. having their assumptions available in future computations on the *same* resolution branch.

3.2 On the *weakening* rule

Two 'structural' rules, *weakening* and *contraction* are used implicitly in classical and intuitionistic logics. Weakening allows *discarding* clauses while contraction allows *duplicating* them.

In Wadler's formulation of linear logic (based on Girard's *Logic of Unity*) they look as follows:

$$\boxed{\frac{\Gamma, [A], [A] \vdash B}{\Gamma, [A] \vdash B}\ Contraction}$$

$$\boxed{\frac{\Gamma \vdash B}{\Gamma, [A] \vdash B}\ Weakening}$$

and do not apply to linear affine (`<A>`) assumptions but only to intuitionistic ones (`[A]`).

The restrictions on the *weakening* rule in linear logic require every (linear) assumption to be eventually used.

On the other hand, *affine* linear logic allows weakening, i.e. proofs might succeed even if some assumptions are left unused.

We found our choice for *affine* linear assumptions practical and not unreasonably restrictive, as for a given linear predicate, *negation as failure* at the *end* of the proof can be used by the programmer to selectively check if an assumption has been actually consumed.

As linear assumptions are consumed on the first use, and their object is guaranteed to exist on the heap within the same AND-branch, no copying is performed and unifications occur on the actual clause. This implies that bindings are shared between the point of definition and the point of use. On the other hand, intuitionistic implications and assumptions follow the usual 'copy-twice' semantics.

4 Assumption Grammars

We will describe in this section how various forms of assumptions can be used for grammar processing conveniently described as an instance of hypothetical resoning.

4.1 Description of the Formalism

Assumption Grammars are logic programs augmented with a) linear and intuitionistic implications scoped over the current continuation, and b) implicit multiple accumulators, useful in particular to make the input and output strings invisible.

As a more convenient notation, we shall use the following equivalences in the remainder of the paper:

```
*A:- assumei(A).
+A:- assumel(A).
-A:- assumed(A).
```

Hidden accumulators allow us to disregard the input and output string arguments, as in DCGs, but with no preprocessing requirement. They are accessible through a set of BinProlog built-ins, allowing us 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/recognize `Phrase`. We refer to [12] for their specification in term of linear assumptions.

For reasons that will become apparent later, we will also define, on top of these builtin assumptions, another type called timeless assumptions:

```
% Assumption:

% the assumption being made was expected by a previous consumption
=X:- -wait(X), !.
% if there is no previous expectation of X, assume it linearly
=X:- +X.

% Consumption:

% uses an assumption, and deletes it if linear
=-X:- -X, !.
% if the assumption has not yet been made,
% adds its expectation as an assumption
=-X: +wait(X).
```

With these definitions, assumptions can be consumed after they are made, or if the program requires them to be consumed at a point in which they have not yet been made, they will be assumed to be "waiting" to be consumed (through "wait(X)"), until they are actually made (at which point the consumption of the expectation of X amounts to the consumption of X itself). Terminal symbols will be noted as: `#word`.

5 A Case Study: Emulating Abductive Logic Programming with Assumptions

5.1 Abductive Logic Programming

An abductive logic program is a triple `<P,A,IC>`, where `P` is a logic program, `A` is a set of *abducible predicates*, and IC is a set of constraints on the abducible predicates in the form of denials, i.e.:

```
<- A1,...,Am.
```

Our constraints may contain variables. If a constraint C contains variables, then it stands for the (possibly infinite) set of its ground instances over the Herbrand universe of P.

An abductive answer to a goal `Q` and `<P,A,IC>` is a conditional answer Q_i :- E_i, such that E_i is a set of instances of abducible predicates, called the *abductive explanation* of Q_i, for which:

- all the constraints in IC are satisfied by each E_i
- Each E_i is minimal in the sense that none of its elements subsumes another element

Note that our definition of abductive logic programming differs from those in the literature in that we do not require groundess of the abduced assumptions or of the constraints. Rather, we ensure minimality of each abductive explanation by requiring that none of their assumptions subsumes another. Obviously, since the answers Q_i are conditional on the explanations E_i, $E_i \cup P$ entails Q_i.

5.2 Computing Abductive Explanations in Assumptive Logic Programming

The following program transformation allows us to compute abductive answers directly, without using a meta-interpreter.

We replace every ocurrence of an abducible predicate A in the right-hand side of a rule in P by a call to `try_abducible(A)`, which tests whether A can be proved from P alone, and if not, assumes it as part of an explanation.

```
add_abduced(A):-
  is_assumed(B),subsumes_check(B,A)->true
;
  % remove the more specific, add the more general one
  -B,
  +A.
```

Add_abduced(A) ensures that an explanation always contains the most general variant of an abducible predicate - as this entails its subsumed instances.

For instance, if a conditional answer requires p(1) and p(X) to be true in order to become an unconditional answer, we shall only keep p(X) as part of the abductive explanation.

5.2.1 Simple Abduction

We can now restate the basic intuition behind the simple rule of abduction, which assumes p as a possible explanation for q given that $p \Rightarrow q$:

```
p => q
q
-------
p
```

With assumptive logic programming, we can express this rule as:

```
q:- p? .
```

In other words: if p has any solution, `p?` succeeds and therefore so does `q` (as in ordinary Prolog). At the end of our computation, we can decide what to conclude from those assumptions that have not been consumed.

5.3 Defeasible Abductive Logic Programming

The ability to examine unconsumed assumptions for further processing implies in particular that we can undo them.

For instance, in a diagnosis problem, the leftover assumptions may correspond to additional symptoms that need to be checked in order to confirm the verdict. If these symptoms are now checked (say, through the system further questioning the user) and found not to be present, we can then undo the corresponding abductions by consuming them, and try some other line of reasoning in our search space.

In a computational linguistics problem, leftover assumptions may be grammar symbols corresponding to words left implicit (e.g., the sentence "Tweedledum proposed, and Twiddledee agreed to, a battle" leaves the object of "proposed" implicit, until it can be reconstructed from the object of "agreed to"). The first subsentence's object will appear as a leftover assumption, to be consumed during the analysis of the second subsentence.

In general, we temporarily make assumptions from incomplete information until we find problem-specific ways of corroborating them. This allows us in particular to undo past abductions. In this sense our framework can be considered a form of defeasible abductive logic programming.

5.4 Conditional Abductive Logic Programming

Using assumptions as a methodology for expressing abduction suggests an interesting extension to abductive logic programming: we can abduce not only predicates, but also clauses, by using scoped assumptions.

Applications are multiple. Of particular interest is inductive logic programming. For instance, we can learn new grammar rules from examples by abducing a grammar rule that has not been stated in the grammar but which would be needed for a given example to be covered by the grammar. Since we make no distinction between clauses and grammar rules, the abduced grammar rule can be represented directly by an abduced clause.

5.5 Types of Abducibles

In the literature, the set of abducible predicates and the set of defined predicates are sometimes disjoint, and sometimes they overlap where abducible predicates can have a partial definition or none at all.

We can assume these sets to be disjoint with no loss of generality. If we want to abduce one more fact, say p(a), about a partially defined predicate p/1, we can simply add an abducible predicate `abduced_p(a)` instead, and complete the (non-abducible) predicate p with the clause:

```
p(X):- abduced_p(a).
```

Another classification of abducibles is suggested by the existence of four types of implication, as in BinProlog: linear, affine linear, intuitionistic and timeless. This suggests another extension to abductive logic programming in which the user can decide whether an abducible predicate is to be consumed exactly once (linear), at most once (affine linear), any number of times (intuitionistic), or independently of when the corresponding assumption has been made (before or after consumption).

Intuitionistic assumptions, however, cannot be used as abducibles as is because by definition they cannot be removed, and we need to be able to remove those assumptions which are found to be inconsistent with the integrity constraints. As we shall see later, we can nevertheless introduce a useful variant of intuitionistic assumption in which the scope can be dynamically narrowed, from its present realm of the entire continuation, into some point defined by the programmer.

It is not impossible but it is quite difficult to use timeless assumptions as abducibles, because the subsumption check would need a complex case analysis.

We will therefore only extend abductive logic programming with two extra types of abducibles, other than the traditional linear one: linear affine and scoped intuitionistic. We next discuss each of our abducibles in turn.

Notice that no declarations are needed for our compiler to distinguish each type, since these are syntactically marked in our abductive logic program.

5.6 Anaphora

We shall now illustrate how assumption grammars can deal with intersentential dependencies through the example of anaphora, in which a given noun phrase in a discourse is referred to in another sentence, e.g. through a pronoun. We refer to the noun phrase and the pronoun in question as entities which co-specify, since they both refer to the same individual of the universe.

As a discourse is processed, the information gleaned from the grammar and the noun phrases as they appear can be temporarily added as hypotheses ranging over the current continuation. Consulting it then reduces to calling the predicate in which this information is stored.

We exemplify the hypothesizing part through the following noun phrase rules:

```
np(X,VP,VP):- proper_name(X),  +specifier(X).
np(X,VP,R):- det(X,NP,VP,R), noun(X-F,NP), +specifier(X-F).

pronoun(X-[masc,sing]):- #he.
pronoun(X-[fem,sing]):- #her.

anaphora(X):- pronoun(X).

noun(X-[fem,sing],woman(X)):- #woman.
```

The linear assumption, `+specifier(X)`, keeps in `X` the noun phrase's relevant information. In the case of a proper name, this is simply the constant representing it plus the agreement features gender and number; in the case of a quantified noun phrase, this is the variable introduced by the quantification, also accompanied by these agreement features.

Potential co-specifiers of an anaphora can then consume the most likely co-specifiers hypothesized (i.e., those agreeing in gender and number), through a third rule for noun phrase:

```
np(X,VP,VP):- anaphora(X), -specifier(X).
```

Semantic agreement can be similarly enforced through the well-known technique of matching syntactic representations of semantic types.

This methodology can of course be extended in order to incorporate subtler criteria. For instance, we can make each pronoun carry, at the end of the analysis, the whole list of its potential referents as a feature. User-defined criteria can then further refine the list of candidate co-specifiers, as in [3].

It is interesting to point out that in order to handle abstract co-specifiers [2], such as events or propositions, all we have to do is to extend the definition so that other parts of a sentence can be identified as possible specifiers as well. For instance, for recognizing "John kicked Sam on Monday" as the co-specifier of "it" in the discourse: "John kicked Sam on Monday. It hurt.", we can simply make the linear assumption that sentences are potential co-specifiers for pronouns of neuter gender.

6 Assumption Grammars and Link Grammars

It is interesting to note that AGs promote both top-down and data driven thinking in the development of a grammar. We employed the latter in our "data-driven" examples, i.e., those with rules in which a terminal symbol is the left-hand side symbol.

We can exploit the data driven thinking mode for AGs in order to emulate another interesting type of grammars: Link grammars [11].

6.1 Link grammars — informal definition and example

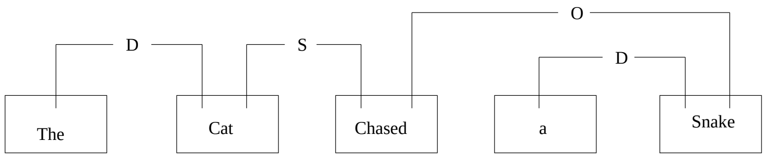
A link grammar consists of a set of terminals ('words') each of which has a *linking requirement*. Planarity (links that do not cross can be drawn over the terminals) and connectivity (all terminal of a recognized phrase can be linked) constraints should be satisfied for each terminal.

The linking requirements of each word are contained in a dictionary. A sample dictionary follows as an illustration:

words	**formulas**
a, the	D+
ran	S-
Mary, John	O- or S+
chased	S- & o+
snake, cat	D- & (O- or S+)

The linking requirement for each word is expressed as a formula involving the operators & and or. The + or - suffix on a connector name indicates the direction (relative to the word being defined) in which the matching connector (if any) must lie.

The following diagram shows how the linking requirements are satisfied in the sentence "The cat chased a snake".



6.2 A sample translation from Link grammars to Assumption Grammars

The translation into AGs is immediate, considering the representation of link grammar dictionaries shown in the previous section. Below, "+" and "-" have our AG meaning of linear implication and consumption respectively, but with this interpretation they happen to do exactly the same job as the link grammar shown above. All we have to do is to transform the + and - suffixes into prefixes, and add weakening for all predicates being assumed:

```
a:- +d.
the:- +d.

mary:- -o; +s.
john:- -o; +s.

chased:- -s, +o.

snake:- -d, (-o ; +s).
cat:- -d, (-o ; +s).

all_consumed:- --s, --o, --d.
s:- (mary,chased,a,snake),all_consumed.
```

Notice that AGs are descriptively more powerful than Link grammars because as in all logic grammars, their symbols can include arguments, through which we can, for instance, dynamically construct sentence representations as a side-effect of parsing. Conceptually, moreover, we can view AGs as enabling a rendering of link grammars in which the higher-order notion of retractable assumption replaces the more procedural notion of "to the right" or "to the left"

7 Related Work

Existing work on Linear Logic based Natural Languages processing [5, 1] 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.

Compared with other Linear (Intuitionistic) Logic based systems like Lolli [6, 7], our constructs are implemented on top of a generic Prolog engine.

We have chosen to allow *weakening* but not *contraction* for linear clauses. Explicit negation as failure applied to facts left over in a proof allows to forbid weakening selectively. We have also chosen to avoid explicit quantifiers, to keep the language as simple as possible. The semantics of our constructs is an instance of the sequent calculus based descriptions of Horn Clause Logic and the more powerful *uniform proof* based systems of [6]. We can see AGs and accumulator processing in general as an even more specific instance of linear operations.

8 Conclusion

We have shown Assumptive Logic Programming to be a powerful formalism for hypothetical reasoning, and exemplified this by showing how to emulate Abductive Logic Programming through assumptions, and by developing its grammatical counterpart, Assumption Grammars.

Assumption Grammars, although theoretically no more powerful than previous logic grammars, have more expressive power in that they permit the specification of rewriting transformations involving components of a string separated by arbitrary strings with the sole resource of intuitionistic and (affine) linear assumption scoped over the current AND continuation. Implementation is immediate through BinProlog's intuitionistic and linear assumptions.

It was surprising to us to discover how directly link grammars could be expressed in AG terms. As well, this discovery motivated us to investigate the data driven mode of AG description, which in itself is another interesting development.

We have presented in a unique framework a set of fairly portable tools for hypothetical reasoning in logic programming languages and used them to specify some previously known techniques, such as Extended DCGs, which have been described in the past only by their implementation.

AGs are useful for writing various programming language processors (compilers, program transformation systems, partial evaluators etc.). They can contribute to the writing of compact and efficient code with very little programming effort.

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 simplifies the use of 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

[1] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, Jerusalem, Israel, 1990. MIT Press.

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

[3] A. Fall, V. Dahl, and P. Tarau. Resolving co-specification in contexts. In *Proc. of IJCAI Workshop on Context in Natural Language Processing*, Montreal, Canada, 1995.

[4] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, (50):1–102, 1987.

[5] J. Hodas. Specifying Filler-Gap Dependency Parsers in a Linear-Logic Programming Language. In Krzysztof Apt, editor, *Logic Programming Proceedings of the Joint International Conference and Symposium on Logic programming*, pages 622–636, Cambridge, Massachusetts London, England, 1992. MIT Press.

[6] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, May 1994. Available as University of Pennsylvania Technical Reports MS-CIS-92-28 or LINC LAB 269.

[7] Joshua S. Hodas and Dale Miller. Logic Programming in a Fragment of Intuitionistic Linear logic. *Journal of Information and Computation*, 110(2):327–365, May 1994.

[8] A. P. Kopylov. Decidability of linear affine logic. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 496–504, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[9] D. Miller and G. Nadathur. Some uses of higher-order logic in computational linguistics. In *24st Annual Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.

[10] D.A. Miller. A logical analysis of modules in logic programming. *J. Logic Programming*, 6(1–2):79–108, 1989.

[11] Daniel D. K. Seator and Davy Temperley. Parsing English with a Link Grammar. Technical report, 1991. CMU-CS-91-196.

[12] Paul Tarau, Veronica Dahl, and Andrew Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In Joxan Jaffar

and Roland H.C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, December 1996. Springer.

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