Backtrackable State with Linear Affine Implication and Assumption Grammars

Paul Tarau[1] Veronica Dahl[2] Andrew Fall[2]

[1] Université de Moncton
Département d'Informatique
Moncton, N.B. Canada E1A 3E9,
tarau@info.umoncton.ca
[2] Logic and Functional Programming Group
Department of Computing Sciences
Simon Fraser University
Burnaby, B.C. Canada V5A 1S6
{veronica,fall}@cs.sfu.ca

Abstract. A general framework of handling state information for logic programming languages on top of backtrackable assumptions (linear affine and intuitionistic implications ranging over the current continuation) is introduced. Assumption Grammars (AGs), a variant of Extended DCGs handling multiple streams without the need of a preprocessing technique, are specified within our framework. Equivalence with DCGs is shown through a translation from AGs to DCGs and through use of an implementation-independent meta-interpreter, customized for handling both DCGs and AGs.
For AGs, efficiency comparable to that of preprocessing based DCG is obtained through a WAM-level implementation which uses space only for non-deterministic execution while collapsing to a form of in-place update in case of deterministic execution. AGs have been fully integrated in BinProlog 5.00 (available by ftp from clement.info.umoncton.ca) and have been used as an alternative to DCGs in various applications.
Not restricted to Prolog, the techniques described in the paper are portable to alternative logic programming languages like Lolli, Lygon and λProlog.
Keywords: state in logic and functional programming, linear affine and intuitionistic implication, accumulator processing, alternative DCG implementations, destructive assignment, logical global variables, continuations.

1 Introduction

It has been recognized since Colmerauer's work on Metamorphosis grammars that definite clauses subsume context-free grammars. 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 inability to support 'multiple streams', although elegantly solved with Peter Van Roy's Extended DCGs [Van89], 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.

In an apparently distinct line of thought, intuitionistic logic and, more recently linear logic [Gir87] have been influential on logic programming and logic grammars [Hod94]. 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.

This brings us to the initial motivation of this work: we wanted to design a set of powerful natural language processing tools to deal with the complex hypothetical reasoning problems which arise, e.g., when dealing with anaphora resolution, relatives, co-ordination etc. The proposed grammars were initially an attempt to deal with the problems of DCGs, while extending their functionality to support multiple streams, as in [Van89]. Surprisingly, the outcome went beyond the intended application domain. A unified approach to handle *backtrackable state information in nondeterministic logic languages*, based on a simplified form of linear affine and intuitionistic implications (assumptions) has emerged.

2 Representing state in logic languages

We will start here with a quick overview of the difficulties of handling efficiently and cleanly state information in declarative languages. The main problem is that, expressing change contradicts some of the basic principles logic (and functional) languages are built on. By definition, 'referential transparency' is lost when a given symbol denotes different objects within the same scope. It has been recognized however, that this has limited impact on *single-threaded* data.

2.1 Re-usability of single-threaded data types

Data having a unique producer and a unique consumer are frequent in declarative programming languages. Work on linear types [Wad91], monads [Wad92] and linear language constructs [Bak92] in functional programming has shown that single-threaded objects are subject to *in-place update* within a reasonably clean semantic framework.

In Prolog, the *hidden 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. Clearly, no practical readability problems occur by collapsing such chains having exactly 2 occurrences of each variable. Arguably, reduced

visual noise will compensate for keeping in mind that implicit state is passed from one literal to another, when this becomes simply: `a :- b,c,d`.

However, 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. Our implementation described in subsection 5.3 solves this problem.

2.2 Scope and state

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*. This is actually the technique used for BinProlog's definition of implication:

```
(C => G) :- assume(Scope,C), G, Scope='$closed'.
```

It ensures, with an appropriate test at *calling time*, that assumption `C` is local to the proof of `G` [3]. Note also that embedded uses require a stack i.e. *asserta*-style discipline for the *assume* operation. Programming with assumptions ranging over the current continuation is exactly the form of hypothetical reasoning described in [TB89].

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] See the actual implementation in file extra.pl of the BinProlog 5.00 distribution, [Tar96].

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`[4] 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` as alternative `X=13` as 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.

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]`).

[4] 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 linear affine implication differs semantically from usual linear implication, it is reasonable to denote it differently.

The restrictions on the *weakening* rule in linear logic require every (linear) assumption to be eventually used. Often, when assumptions range over the current continuation, this requirement is too strong, except for the well-known situation of handling *relative clauses* through the use of *gaps* [Hod92]. 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. It is also possible to check through the addition of a low-level primitive, that at a given point, the set of all affine linear assumptions is empty.

Implicit sharing/copying conventions Although intuitionistic logic based systems like λProlog and linear logic implementations usually support quantification with the benefit of additional expressiveness, we have chosen (in compliance with the usual Horn Clause convention) to avoid explicit quantifications, for reasons of conceptual parsimony and simplicity of implementation on top of a generic Prolog compiler.

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 Expressiveness of assumptions

We will show the expressiveness of affine linear assumptions through an example a variant of which has been independently discovered by the creators of the logic language Lygon [WH95].

Loop-avoidance in graph walking with linear implication It is unexpectedly easy to write a linear implication based graph walking program. It will avoid falling in a loop simply because linear implication (`-:`) assumes facts that are usable only once (i.e. consumed upon their successful unification with a goal).

```
path(X,X,[X]).
path(X,Z,[X|Xs]):-linked(X,Y),path(Y,Z,Xs).

linked(X,Y):-c(X,Ys),member(Y,Ys).

start(Xs):-
  c(1,[2,3])-:c(2,[1,4])-:c(3,[1,5])-:c(4,[1,5])-:
  path(1,5,Xs).
```

By executing `?-start(Xs)`, we will avoid loops like `1-2-1 and 1-2-4-1` and obtain the expected paths:

```
Xs=[1,2,4,5];
Xs=[1,3,5]
```

[htbp]

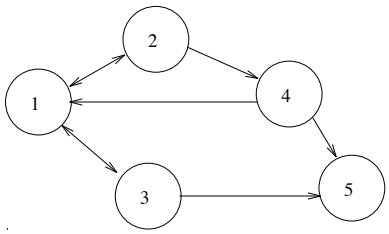

Fig. 1. The graph with loops described by `c/2`

Note that the adjacency list representation of the graph in fig. 1 ensures that each node represented as a linear assumption `c/2` becomes unavailable, once visited. This makes our example simpler than the similar program given in the Lygon distribution. Note also that without *weakening* we would have a hamiltonian walk [5].

5 Assumption Grammars

BinProlog's Assumption Grammars (shortly AGs) can be easily specified in terms of `assumel/1` by attaching to each DCG stream a set of linear facts (see [Tar96]).

In terms of generality, they are a superset of DCG grammars covering an essential part of the functionality of Peter Van Roy's Extended DCGs [Van89]. No preprocessing is involved in the implementation of AGs, as WAM-level backtrackable destructive assignment and global state for each DCG stream is used.

5.1 Translation semantics for AGs

At first sight it might seem that AGs lack the 'logical' semantics of ordinary DCGs which are usually implemented by translation to side-effect free Prolog

[5] Which, unfortunately, is NP-complete. This is another reason why we have chosen to leave to the programmer the task to forbid *weakening* only when really needed. It is also interesting to notice, in this context, that propositional (multiplicative, additive, exponential) affine linear logic is decidable [Kop95] while its linear logic counterpart is not.

code. To keep things simple let us restrict ourself to a unique AG stream. By replacing each occurence of `'#'(X)` with `[X]` and each occurrence of `:-` by `-->` in a each 'AG grammar rule' its semantics will be unchanged. For fully emulating DCGs, one can use directly `-->` in a AG based program, and replace the DCG preprocessor with something like `expand_term((A-->B),(A:-B))`:

```
x:- y,#a,z.        x --> y,[a],z.
y:- #b,#c.  ==>    y --> [b],[c].   ==> Prolog
z:- #d,#e.         z --> [d],[e].
```

Alternatively, the reverse translation is usable as a 'DCG implementation', with `#/1` a WAM-level built-in.

5.2 A generic meta-interpreter for DCGs and AGs

We will show by writing down a basic meta-interpreter which can be easily customized for either DCGs or AGs, that the two are basically alternative implementations of the same abstract algorithm.

```
interp((A,B),S1,S3):-!,interp(A,S1,S2),interp(B,S2,S3).
interp(T,S1,S2):-terminal(T,X),!,connect(X,S1,S2).
interp(E,S,S):-empty(E),!.
interp({Goal},S,S):-!,Goal.
interp(H,S1,S2):-rule(H,B),interp(B,S1,S2).
```

DCG instantiation

```
rule(H,B):- '-->'(H,B).              empty([]).
connect(X,S1,S2):- 'C'(S1,X,S2).     terminal([X],X).
```

AG instantiation

```
rule(H,B):-clause(H,B).          empty(true).
terminal(#(X),X).                connect(X,[X|S2],S2).
```

5.3 High performance implementation of AGs

For reasons of efficiency BinProlog's AGs have been implemented in C and are accessible through a set of builtins [Tar96]. AGs do not need to represent the chain of existential variables (heap-represented on most WAMs) introduced by the usual DCG and EDCG transformations. Instead, backtrackable destructive assignment implemented with *value trailing*[6] is used. The builtin `#/1` working as the 'connect' relation `'C'/3`, in DCGs, consumes trail-space only when a nondeterministic situation (choice-point) arises. This is achieved by address-comparison with the top of the heap, saved in the choice-point. By 'stamping'

[6] Value-trailing consists in pushing both the address of the variable and its value on the trail.

the heap with an extra cell inserted in the reference chain to the value-trailed objects, further attempts to trail the same address will see it as being *above* the last choice point. *This actually results in constant heap/trail use for each chain, only when a choice point is created, and no heap/trail use otherwise.*

This is complemented with a very efficient, 'if-less' *un-trailing* operation based on indirect address calculation. Despite the extra run-time effort, as `#/1` actually uses a specialized instance of `setarg/3`, the overall performance of this *run-time* technique is fairly close to the *static* transformation based approach, even for plain DCGs, while offering multiple-stream functionality. We have also given emulated and native SICStus 2.1_9 figures (DCG-emSP and DCG-natSP) to show that performance is measured w.r.t a fairly efficient DCG processor (DCG-emBP).

compiler	DCG-emSP	DCG-natSP	DCG-emBP	AG-emBP
expr(4) (ms):	230	110	320	340
expr(5) (ms):	1930	840	2460	2980
expr(6) (ms):	18910	8490	23410	25850

Table 1. DCGs vs. AGs

The table 1 shows the comparative speed of AGs vs. DCGs on parsing all the well formed expressions of length N=4,5,6, for an arithmetic expression grammar.

5.4 Portability: Assumption Grammars in Life

Porting Assumption Grammars to a language which has global variables and backtrackable destructive assignment is easy. Here is the code for Wild-Life. Extending this to multiple dcg streams is straightforward.

```
global(dcg_stream)?

dcg_def(Xs) :- dcg_stream <- s(Xs).
dcg_val(Xs) :- dcg_stream = s(Xs).
dcg_connect(X) :- dcg_stream = s([X|Xs]), dcg_stream <- s(Xs).
dcg_phrase(Axiom,Xs) :- dcg_def(Xs), Axiom, dcg_val([]).
```

6 Related work

Compared with other Linear (Intuitionistic) Logic based systems like Lolli [Hod94, HM94], 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 [Hod94]. We can see AGs and accumulator processing in general as an even more specific instance of linear operations.

To avoid passing extra arguments to predicates which do not use them, the accumulator preprocessor of `Wild-Life 1.01` (based on EDCGs [Van89]) requires `pred_info` declarations saying which predicates make use of which accumulators. An advantage of Assumption Grammars, compared with DCGs and EDCGs is that no preprocessing potentially hiding the programmer's intent at source level is required. This becomes important for easier debugging and direct use of meta-programming constructs.

7 Conclusion

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. Our full WAM-level description of AGs [TDF95] can be easily replicated in any Prolog system.

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. As shown in [FTD95] they can be successfully used also for complex natural language processing systems.

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 use of DCGs while offering more powerful facilities in the form of hypothetical assumptions and multiple accumulators (EDCGs). This also reduces the need for explicitly imperative constructs like *assert* and *retract* in logic programming languages.

More work is needed for the formalization of the subset of linear and intuitionistic logic we have implemented and described in the paper.

On the implementation side, further research is needed on inferring more, statically, about particular instances of linear and intuitionistic assumptions, which would allow very small overhead over classical statically compiled code. The combination of AGs and linear/intuitionistic assumptions is a practical basis for building semantically clean object oriented extensions on top of Prolog.

Acknowledgment

Paul Tarau thanks for support from NSERC (research grant OGP0107411), and from the FESR of the Université de Moncton. Veronica Dahl thanks for support from the Canadian National Science and Engineering Research Council (research

grant 31-611024). Special thanks to the anonymous referee pointing out the relationships between our work and *affine linear logic*.

References

[Bak92] H. Baker. Lively linear lisp–'look ma, no garbage!'. *ACM Sigplan Notices*, 27(8):89–98, August 1992.

[FTD95] Andrew Fall, Paul Tarau, and Veronica Dahl. Natural Language Processing with Hypothetical Assumption Grammars and Sparse Term Taxonomies. Technical Report 95-3, Département d'Informatique, Université de Moncton, April 1995. Available by ftp from *clement.info.umoncton.ca*.

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, (50):1–102, 1987.

[HM94] 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.

[Hod92] 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.

[Hod94] 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.

[Kop95] 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.

[Tar96] Paul Tarau. BinProlog 5.25 User Guide. Technical Report 96-1, Département d'Informatique, Université de Moncton, April 1996. Available from *http://clement.info.umoncton.ca/BinProlog*.

[TB89] Paul Tarau and Michel Boyer. Prolog Meta-Programming with Soft Databases. In Harvey Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 365–382. MIT Press, 1989.

[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.

[Wad91] P. Wadler. Is there a use for linear logic? *ACM/IFIP PEPM Symposium*, June 1991.

[Wad92] Philip Wadler. The essence of functional programming. In *ACM Symposium POPL'92*, pages 1–15. ACM Press, 1992.

[WH95] Michael Winikoff and James Harland. Implementing the linear logic programming language Lygon. In John Lloyd, editor, *International Logic Programming Symposium*, pages 66–80, Portland, Oregon, December 1995. MIT Press.

Appendix: A specification of AGs as linear affine assumptions

```
% creates and initializes a named 'Extended DCG' stream
ag_def(Name,Xs):-assumed(ag_state(Name,_)),!,assumel(ag_state(Name,Xs)).
ag_def(Name,Xs):-assumel(ag_state(Name,Xs)).

% unifies with the current state of a named 'DCG' stream
ag_val(Name,Xs):-ag_state(Name,Xs),assumel(ag_state(Name,Xs)).

% equivalent of the 'C'/3 step in Prolog
ag_connect(Name,X):-ag_state(Name,[X|Xs]),assumel(ag_state(Name,Xs)).

% EDCG equivalent of phrase/3 in Prolog
ag_phrase(Name,Axiom,Xs,End):-ag_def(Name,Xs),Axiom,ag_val(Name,End).

% file I/O inspired metaphors for switching between streams
ag_tell(Name):-assumed(ag_name(_)),!,assumel(ag_name(Name)).
ag_tell(Name):-assumel(ag_name(Name)).

ag_telling(Name):-assumed(ag_name(Name)),assumel(ag_name(Name)).
ag_telling(Name):-ag_default(Name).

% projection of previous operations on default DCG stream
ag_default(1).

ag_def(Xs):-ag_telling(Name),!,ag_def(Name,Xs).
ag_def(Xs):-ag_default(Name),ag_tell(Name),ag_def(Name,Xs).

ag_val(Xs):-ag_telling(Name),ag_val(Name,Xs).
ag_connect(X):-ag_telling(Name),ag_connect(Name,X).

ag_phrase(Axiom,Xs,End):-
   ag_telling(Name),ag_def(Xs),ag_phrase(Name,Axiom,Xs,End).

% syntactic sugar for 'connect' relation, in BinProlog 5.00 notation
#W:-ag_connect(W).

% example
axiom:-ng,v.   ng:-a,n.    a:- #the.      a:- #a.
n:- #cat.      n:- #dog.   v:- #walks.    v:- #sleeps.

?-ag_phrase(axiom,Xs,[]).

Xs=[the,cat,walks];
Xs=[the,cat,sleeps];
..............
Xs=[a,dog,sleeps]
```

This article was processed using the LaTeX macro package with LLNCS style