Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars

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

A set of executable specifications and efficient implementations of *backtrackable state* persisting over the current AND-continuation is investigated.

At specification level, our primitive operations are linear and intuitionistic implications, having as consequent the current continuation. On top of them, we introduce a form of hypothetical assumptions which use no explicit quantifiers and have an easy and efficient implementation on top of logic programming systems featuring backtrackable destructive assignment, global variables.

A variant of Extended DCGs handling multiple streams without the need of a preprocessing technique, Hidden Accumulator Grammars (HAGs), are specified in terms of linear assumptions. For HAGs, efficiency comparable to that of preprocessing techniques is obtained through a WAM-level implementation of backtrackable destructive assignment, supporting non-deterministic execution based on *value-trailing*, while collapsing to a form of in-place update in case of deterministic execution.

When more precise information about the location where linear assumptions are be *used* is available, either HAGs or a continuation based representation (*multi-headed* clauses) is used to avoid a performance penalty due to dynamic implementation.

Not restricted to Prolog, the techniques described in the paper are portable to alternative logic programming languages like Life, Lolli and λProlog.

Keywords: state in logic and functional programming, linear and intuitionistic implication, accumulator processing, alternative DCG implementations, destructive assignment, logical global variables, continuations.

1 Introduction

Intuitionistic logic and, more recently linear logic [Gir87] have been influential on logic programming systems and the theory of logic programming [MN86, Mil89b, HM91, Hod93]. The result is not only a better understanding[1] of their proof-theoretical characteristics but also a growing awareness on the practical benefits of integrating them in conventional Prolog systems.

In contrast to previous papers, we will adopt here 'the practitioner's perspective', with emphasis on immediate usability of working code in problem solving tasks. The concepts we have borrowed from the (impressive!) previous research have been selected for their comparative expressiveness, and not without seeking various theoretical and practical alternatives.

The initial motivation of this work was 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. [2].

As we became aware during the process of implementing various tools, the outcome goes beyond the intended application domain to a unified approach to handle backtrackable state information in nondeterministic declarative languages.

The paper is organized as follows: section 2 revisits the handling of state information in declarative languages. Section 3 presents our linear and intuitionistic implication based hypothetical reasoning tools and section 4 shows their usefulness in a few 'proof-of-concept' applications. In section 5, a type of grammars based on global objects with backtrackable state (HAGs) are specified in terms of linear assumptions, followed by the overview of their efficient WAM-level implementation. Section 6 shows how first-order AND-continuations are used for implementing special cases of linear assumptions. Section 7 discusses related work and similar facilities in logic languages like Life. Future work is discussed in section 8 and our conclusions are given in section 9. Two appendices contain additional information on the actual WAM-implementation of HAGs and an example of higher-order objects with hypothetical state, implemented in our framework.

2 State information in declarative languages

We will start here with a quick overview of the problem of handling efficiently and cleanly state information in logical and functional languages.

The basic problem is that in the general case, expressing change contradicts some of the basic principles functional and logic languages are built on. We will revisit some interesting special cases, where the problem has been either solved or at least clearly stated. We will also give, at this stage, a few quick hints about some of the contributions we will describe later in more detail.

[1] For instance *uniform proofs* [MNPS91]) indicate that a formalism is likely to be computationally interesting.

[2] We refer to [TDF95] for the applications of our techniques to natural language processing.

2.1 Re-usability of single-threaded data types

Objects having a unique producer and a unique consumer are frequent in declarative programming languages.

Work on linear types [Wad91], monads [Wad93] and linear language constructs [Bak92a] 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.

However, in the presence of backtracking, previous values must be kept for use by alternative branches. This situation conflicts with possibility of reuse, and makes single-threaded objects more complex (but, arguably more powerful) in non-deterministic LP languages than in committed choice or functional languages.

Our solution will be to reuse the space for single-threaded chains of variables while in a deterministic branch and value-trail (i.e. keep the address of the variable and its value on the trail-stack) otherwise. Clearly, for a single-threaded object, one such value-trailing operation by choice-point is enough. We will describe how this is done in BinProlog [Tar95] using a simple address comparison technique in subsection 5.2.

2.2 Scope and state

From a very general perspective, state is always local. Seen from the kernel of the operating system, whatever happens inside the address space of a process is local to the process [3].

Outside of these protective environments, however, talking exclusively about local state is almost an oxymoron. The reason we need state is because *there are* things that do not change locally, *and* we still have to pass them around as arguments. This usually conflicts with a programmer's healthy 'minimal action' instinct.

Moreover, in logic programming languages, instantiation of a local variable is actually propagated globally in often unpredictable ways, through unification. Note that lazy functional languages or constraint solving systems also trigger delayed actions which are difficult to predict operationally [4].

Therefore, both declarative paradigms are already stateful, at least when we look close enough to their actual execution mechanisms. This suggests, that some form of explicit state manipulation will not necessarily make them worse, especially when in actually helps to hide unwanted information.

It is important, for a given programming paradigm to identify the appropriate concept of state. For non-deterministic logic programming languages like Prolog, the natural scope for the state of an object is the current AND-continuation [5], as

[3] This is 'kindly' enforced with "Segmentation fault" messages.

[4] That's why it would be hard to conceive a language like Miranda or Haskell without referential transparency.

[5] This is different from what's happening in functional languages like Haskell where, in a deterministic (although lazy evaluation enabled) framework, elegant unified solutions have been described in terms of monads and continuations, and 'imperative functional programming' is used (with relative impunity) for arrays, I/O processing, etc.

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. This will be the main reason why our linear and intuitionistic assumptions are scoped over the current continuation.

2.3 Scope shunting

Suppose that a backtrackable *assume* primitive is efficiently implemented in a given language, where assumptions range over the future of the current AND branch. Note that it is easy to shunt down 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` [6].

2.4 State as a resource

Another important parameter besides scoping is definite or indefinite usability of an assumption. It is quite familiar, with the advent of Linear Logic [Gir87], to see axioms and theorems as resources with a possibly limited number of uses.

Let us remark that this is a trivial property of physical objects which has been successfully abstracted in the world of computers, where *copying* an object costs the same or less than *moving* it to another destination. The paradoxical consequence of the 'thermodynamics' [Bak92b] of this unlimited copying framework, is that in a very essential way, safe parallel execution with locking of incompletely copied objects cannot be directly expressed.

Coordination languages like Carrierro and Gelernter's Linda [CG89] will therefore *move* (and **not** *copy*) objects between a tuple space called *blackboard* and individual processes to ensure not only information exchange but also synchronisation with a very limited number of simple primitive operations:

- `out/1` copies an object to the blackboard
- `in/1` moves a 'matching' object back in process space or blocks until it is available
- `rd/1` checks availability of a matching object on the blackboard and signals failure otherwise.

[6] See the actual implementation in file extra.pl of the BinProlog 3.30 distribution, [Tar95].

3 Hypothetical reasoning with linear and intuitionistic logic

3.1 Assumed code, intutionistic and linear implication

Although the Linda framework is already implemented as an extension to BinProlog [DBT93], it does not provide hypothetical reasoning facilities due to the non-backtrackable nature of the blackboard operations which, like `assert` and `retract`, would produce imperative state changes.

We have implemented some new hypothetical reasoning facilities on top of the BinProlog abstract machine which already had backtrackable destructive assignment and a form of logical global variables. We will give a short description of the primitive operations and point out some of the differences with other 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.

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. `?-assumei(a(13)),a(X),a(Y)` will succeed.

Linear assumel/1 adds a clause usable *at most once* in subsequent proofs. This assumption also vanishes on backtracking. Its scoped version `Clause -: Goal`[7] 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. `?-assumel(a(13)),a(X),a(Y)` will fail.

The user can freely mix linear and intutionistic clauses and implications for the same predicate, as in:

```
?-a(10) -: a(11) => a(12) -: a(13) => (a(X),a(X)).
X=11;
X=13;
no
```

In this example, `a(10)` and `a(12)` are *consumed* after their first use while `a(11)` and `a(13)` are reusable indefinitely.

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

Achieving this at source level by explicit continuation handling is possible, although somewhat complex and inefficient [8]. Roughly speaking, `assumei/1` and `assumel/1` could have been defined as:

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

[8] Inefficiency comes from the fact that it would imply dispatching and interpreting the current continuation. Even on systems like BinProlog where the current continuation is a first class object, it would incur unnecessary overhead to 'un-binarize' it, i.e. to undo the effect of BinProlog's preprocessing transformation.

```
assumei(Clause):-
  get_cc(CurrentContinuation),
  Clause => CurrentContinuation.

assumei(Clause):-
  get_cc(CurrentContinuation),
  Clause -: CurrentContinuation.
```

We have prefered to go the other way around and implement `assumei/1` and `assumel/1` as primitives, while implementing the scoped implications on top of them.

3.2 On the *weakening* rule

The *weakening* rule in linear logic requires every assumption to be eventually used.

Often, 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 [Hod92].

Therefore, BinProlog's linear implication will succeed even if not all the assumptions are consumed (*weakening* is allowed), while in systems like Lolli their 'consumption' is a strong requirement, i.e. it is enforced for success.

We found our choice 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 check if the 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 linear assumptions is empty.

3.3 Overriding

Assumed predicates will override similarly named dynamic predicates which in turn will override compiled ones. Note that overriding is done at *predicate*, not at *clause* level. This allows overriding a set of default predicate definitions at runtime with the overall result of an implementation of 'open programs'. Note that overriding also offers the ability to define 'parametric modules'.

3.4 Implicit 'quantification' 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, mostly for simplicity of implementation on top of a generic Prolog compiler [9].

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 can be seen (through the usual

[9] And also because they create unrealistic expectations on the programmer's side, in the same way as speech or hand-writing recognition create them for casual computer users, who expect the 'real thing' after seeing a look-alike instance. We remind the reader not convinced about the advantages of implicit quantification, that *functions* are nothing but implicitly quantified relations and and that our declarative cousins - functional programming languages -, and more generally, working mathematicians, do not consider this such a bad idea after all :-).

language abuse) as implicit 'existential quantification'. On the other hand, intuitionistic implications and assumptions follow the usual 'copy-twice' semantics [10] of assert and retract, and can be seen as being 'universally quantified' [11].

3.5 Horn Clause Translation Semantics

We have implemented `assumei/1` and `assumel/1` in BinProlog entirely at source level [12], with backtrackable destructive assignement used only for efficiency reasons. Clearly, our implicit quantification rules ensure that linear assumptions can be given a Horn Clause 'translation semantics' in terms of DCG-style accumulator processing, by keeping the list of assumed clauses in a chain of DCG arguments, following the preprocessing technique described in [TB89]. On the other hand, intuitionistic assumptions can be expressed at source level in terms of Horn Clause Logic with the addition of a copying primitive (available as `copy_term/2` in ISO-Prolog).

4 Expressiveness

We will show in this section that our proposed extensions are able to express elegantly constructs which have been found useful in the past.

4.1 Linear Assumptions and the Linda model

The basic transactions of the Linda model [CG89] (`in/1` and `out/1`) correspond closely to our assumptions scoped over the current continuation.

Let us suppose that our assumptions are constrained to be ground facts and that the operations on the Linda blackboard are intended to be backtrackable. Then `assumel/1` is just the usual `out/1` Linda operation and calling the linearly assumed fact is just the usual `in/1` operation!

This shows that Linear Logic basically subsumes the Linda framework and we can in principle build our operations on top of existing Linda systems, while using the blocking/non-blocking variants of the `in/1` operation for co-ordination between parallel threads.

The third Linda operation, `rd/1` can be expressed as `rd(X):-X,assumel(X)`.

4.2 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).
```

[10] This is not strictly needed for implementation reasons, (copying on use would suffice) as the backtrackable nature of intuitionistic implications ensures that their assumptions can be safely heap-represented. Exploring implicitly existentially quantified intuitionistic assumptions is still an interesting alternative.

[11] This should not be confused with the 'new constant' based implementation of universal quantifiers in λProlog [Mil91a, Mil91b].

[12] See file `extra.pl` of the distribution on *clement.info.umoncton.ca*.

```
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 the loop `1-2-1` and obtain the expected paths:

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

Note that the adjacency list representation ensures that each node represented as a linear assumption `c/2` becomes unavailable, once visited. Note also that enforcing *weakening* would give a hamiltonian walk [13].

4.3 Language enhancements based on linear implication

Linear implication can be used to quickly add some language enhancements to a Prolog system. For instance, to add a `switch` ... `case` statement one can write simply:

```
switch(Selector,Body):-
  case(Selector) -: Body.

default:-case(_).

test(X):-
  switch(X, (
     case(1)->write(one) ;
     case(2)->write(two) ;
     default->write(unexpected(X))
  ))
 ,nl.
```

Clearly, this is a very compact source-level implementation for a `switch` construct, useful at least as a specification. Note however, that this should be done with macro expansion in a real implementation, to ensure constant dispatching time on the `case` label. We have chosen *default/0* to consume the assumption `case/1` so that this should also work when *weakening* is enforced by the implementation (as it is the case in the Lolli system).

We refer to Appendix II for an implementation in our framework of backtrackable 'higher-order' state in the form 'replicating' objects.

5 Hidden Accumulator Grammars

5.1 A specification in terms of linear assumptions

With `assumel/1` linear assumptions scoping over the current continuation, we can easily specify a superset of DCG grammars practically equivalent to Peter

[13] Which, unfortunately, is NP-complete. This is another reason why we have chosen to leave to the programmer the task to enforce *weakening* only when really needed.

Van Roy's Extended DCGs [Van89] and their Life [AKP91] counterparts. We call them *Hidden Accumulator Grammars* as no preprocessing will be involved in their implementation. It turns out that the technique has the advantage of 'meta-programming for free' (without the expensive phrase/3, doing either meta-interpretation or on-the-fly DCG-expansion), it allows source level debugging, and it can be made more space and time efficient than the usual preprocessing based implementation. As in the case of EDCGs, their best use is for writing a Prolog compiler (they can contribute to the writing of compact and efficient code with very little programming effort) and, as shown in [TDF95], also for complex natural language processing systems.

Hidden Accumulator Grammars (shortly HAGs) are specified as follows:

```
% creates and initializes a named 'DCG' stream
l_def(Name,Xs):-l_dcg(Name,_),!,assumel(l_dcg(Name,Xs)).
l_def(Name,Xs):-assumel(l_dcg(Name,Xs)).

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

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

% equivalent of phrase/3 in Prolog
l_phrase(Name,Axiom,Xs):-l_def(Name,Xs),Axiom,l_val(Name,[]).

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

l_telling(Name):-l_name(Name),assumel(l_name(Name)).
l_telling(Name):-l_default(Name).

% projection of previous operations on default DCG stream
l_def(Xs):-l_telling(Name),!,l_def(Name,Xs).
l_def(Xs):-l_default(Name),l_tell(Name),l_def(Name,Xs).

l_val(Xs):-l_telling(Name),l_val(Name,Xs).

l_connect(X):-l_telling(Name),l_connect(Name,X).

l_phrase(Axiom,Xs):-l_telling(Name),l_def(Xs),l_phrase(Name,Axiom,Xs).

l_default(1).

% syntactic sugar for 'connect' relation
{W}:-l_connect(W).

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

```
go:-l_phrase(axiom,Xs),write(Xs),nl,fail.
```

5.2 High performance implementation of HAGs

For reasons of efficiency (i.e. to equal or beat preprocessor based DCGs in terms of both space and time) BinProlog's HAGs have been implemented in C (see Appendix I) and are accessible through the following set of builtins:

```
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  % focusses on a given DCG stream
dcg_telling/1 % returns the number of the current DCGs stream

% connect operation: normally macro-expanded
'#'(Word):-dcg_connect(Word).

% example: ?-dcg_phrase(1,(#a,#b,#c),X).
dcg_phrase(DcgStream,Axiom,Phrase):-
  dcg_telling(X),dcg_tell(DcgStream),
    dcg_def(Phrase),
      Axiom,
    dcg_val([]),
  dcg_tell(X).
```

Starting with BinProlog 3.36 `dcg_tell/1` is backtrackable, mostly like the `lval/3` used to handle global logical variables [14]. This makes HAGs fully equivalent to a source-level specification which would add chains of DCG-variables as pairs of extra arguments for each stream. Note that this functionality is obtained despite the fact that we do not require the (relatively tedious) declarations needed in the original EDCGs and in Wild-Life, which specify for each predicate the accumulators they make use of.

Hidden Accumulator Grammars consume no heap as they do not generate the existential variables introduced by the usual DCG transformation. Instead, backtrackable destructive assigment implemented with *value trailing*[15] is used. The builtin `dcg_connect/1` can be seen as using a 'partially-evaluated' version of BinProlog's `setarg/3`. It 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. With cooperation from builtins which want to benefit from the technique by 'stamping' 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 is complemented with a very efficient, if-less *un-trailing* operation based on indirect address calculation. As BinProlog's `setarg/3` is already about 2-3 times faster than the one in SICStus and `dcg_connect/1` actually uses a specialized instance of `setarg/3`, the overall performance of this *run-time* technique is superior to the static, transformation based approach. Besides their space efficiency, HAGs are also usually faster than their preprocessor based equivalents, while offering multiple-stream functionality.

[14] More precisely 2-keyed hashing based global 'properties'.

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

5.3 Portability

5.3.1 Hidden Accumulator Grammars in Life

Porting Hidden Accumulator Grammars to a language which has global variables and backtrackable destructive assignment is easy. Here is the code for Wild-Life [AKP91].

```
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([]).
```

Extending this to multiple dcg_streams is straightforward. The facility complements preprocessor based accumulators already existing in Life.

5.3.2 Hidden Accumulator Datalog Grammars

Prolog's DCG list-based implementation can also be replaced with an assertional representation as in Datalog grammars [DTH94, DTMP95, DT95, BVG95], as the following example shows.

```
{W}:-dcg_val(N),'D'(W,N,NewN),dcg_def(NewN).

dlg_phrase(Axiom,End):-dcg_def(0),Axiom,dcg_val(End).

x:-ng,v.     ng:-a,n.
a:-{the}.    a:-{a}.
n:-{cat}.    n:-{dog}.
v:-{walks}.  v:-{sleeps}.

'D'(a,0,1).
'D'(cat,1,2).
'D'(walks,2,3).

?-dlg_phrase(x,N).
```

We see here a strong argument for not standardizing DCGs in the forthcoming ISO-Prolog, because it is now clear that they are only a possible, one-stream *implementation* of a more general construct. The complexities of interaction between the DCG program transformer, `phrase` and builtins like CUT are, by the way, entirely avoided with HAGs and their DLG counterparts.

5.3.3 Files as sets of Datalog grammar terminals

It is convenient (and often mandatory) for large-sized natural language processing systems to be able to work directly on files.

By working directly on the file position pointer, our Hidden Accumulator based DLGs can easily handle grammars working on an input sequence of a practically arbitrary size.

Let's note that instead of

```
'D'(elephant,3,4)
```

we can refer to something as `'D'(elephant,FilePos1,FilePos2)`, corresponding to the actual position in a file of the word `elephant` the grammar expects.

As most Prolog systems have direct file-positioning primitives, with this representation we can obtain a logic grammar working directly on a file without using a list representation.

This is also useful in case of incremental processing of the content of the file, when only the segment between 2 file-positions is processed.

6 Continuation grammars

As continuations are first order objects in BinProlog, they can be actually used for efficiently implementing special instances of intuitionistic assumptions.

We refer to [TD94] for their use as *continuation grammars* which successfully replace push-back lists in DCGs through the use of a multi-headed clause mechanism. We will only give here an example showing how an *implication* based programming style can be emulated in terms of *continuation manipulation*, when the precise location of the *use* of an assumption is predictable.

One of Miller's motivating examples for (intuitionistic) implication in λProlog [16] [Mil89a], is the `reverse` predicate.

```
reverse(L,K) :-
  all rv\( ( rv([],K),
             all X,N,M\ (rv([X|N],M) :- rv(N,[X|M]))
           ) => rv(L,[])
         ).
```

We rewrite it here (in BinProlog syntax) with the (more efficient) use of linear implication.

```
reverse(Xs,Ys):-
  result(Ys) -: rev(Xs,[]).

rev([],Ys):-result(Ys).
rev([X|Xs],Ys):-rev(Xs,[X|Ys]).
```

Our example will already run at least one order of magnitude faster because the recursive clause (which dominates execution time) is statically known. Note also the convenience and flexibility of our implicit quantification rules.

By using the *multi-headed clauses* of [TD94] which allow a (relatively) high level manipulation of the current continuation we can write a such a reverse predicate as follows:

```
  reverse(Xs,Ys):-rev(Xs,[]),result(Ys).

    rev([],Ys), result(Ys).
    rev([X|Xs],Ys):-rev(Xs,[X|Ys]).
```

[16] which is, by the way, a motivating example also for Andreoli and Pareschi's Linear Objects, [AP90].

which gives after binarization:

```
reverse(Xs,Ys,Cont):-rev(Xs,[],result(Ys,Cont)).

  rev([],Ys, result(Ys,Cont)) :- true(Cont).
  rev([X|Xs],Ys,Cont):-rev(Xs,[X|Ys],Cont).
```

with a suitable definition for `true/1` as:

```
  true(C):-C.
```

and with a clause like

```
  reverse(Xs,Ys):-reverse(Xs,Ys,true).
```

as interface.

Further speed-up comes from the fact that no dynamic clause creation/search is needed for the basic case of the recursive predicate `rev/2`. We have measured that this predicate executes exactly as fast as its equivalent accumulator based Prolog version, which means that when this transformation is possible because of the *known* place where linear assumptions will be used, our continuation based implementation of linear implication adds no overhead compared with static code.

7 Related work

Compared with other Linear (Intuitionistic) Logic based systems like Lolli [Hod93], our approach is implemented on top of a generic Prolog engine, with a deliberate omission of *weakening* rule enforcement and explicit quantifiers. Accumulators are seen here as an even more specific instance of our already linear operations.

Accumulators have been invented to support uniform operation on single-threaded stateful data, which exhibits an associative or *monadic* structure. They generalize Prolog's DCGs which carry around information on the state of a list representing the string generated or recognized by a grammar. Accumulators have been implemented in the original proposal [Van89] and in the Life system [AKP93] through a preprocessor which adds them as extra arguments at source level, while our proposal handles the hidden arguments directly as state of global objects, subject to backtrackable destructive assignment. Multiple accumulators are naturally needed in compiler writing, corresponding to the multiple streams of information (input sequence, syntactic structure construction, error position, symbol table construction, instructions of intermediate code etc.). Similar multiple streams arise in natural language processing with various kinds of logic grammars.

Preprocessor based approaches like the Extended DCGs [Van89] and Soft Databases [TB89] are based on a program transformation which adds extra arguments and deals with their state using a set of Abstract Data Type operations. This state can be seen as data-only [Van89] or as a form of dynamic code, appropriate for hypothetical reasoning [TB89]. In Life [AKP91] accumulators have an additional object-oriented flavor: an arbitrary method can be executed as the accumulator's state advances. This action is backtrackable, as is the advancement in the input list in the case of ordinary DCGs. This more general feature is expressed in our framework directly by implementing objects on top of intuitionistic

and linear assumptions, although HAGs can be used to do it more efficiently. To avoid passing extra arguments to predicates which do not use them, the accumulator preprocessor of `Wild-Life 1.01` requires `pred_info` declarations saying which predicates make use of which accumulators. For instance,

```
pred_info([p,q],[a])?
```

will indicate that predicates p and q will have extra features for keeping accumulator states, i.e.:

```
p:--q,p?
```

and it will be translated as:

```
p(in_a=>S1,out_a=>S3):-
    q(in_a=>S1,out_a=>S2),
    p(in_p=>S2,out_p=>S3).
```

The generalization over DCGs is that multiple extra feature pairs can be defined.

The `acc_info` declarations complement `pred_info` declarations by describing what kind of methods are triggered when the state of the accumulator pair advances.

The main advantage of our Hidden Accumulator Grammars technique is that no declarations and no preprocessing potentially hiding the programmer's intent at source level are required. This becomes important for easier debugging and use of meta-programming constructs [17].

Working with continuation [BR93] is usual in systems like λProlog-MALI [BR92]. We think however that our primitives `assumel/1` and `assumei/1`, explicitly designed as scoped over the current continuation, allow us to express directly constructions available only as expert-programmer tricks otherwise.

8 Future work

Once familiarized with the idea that clauses are resources, it becomes clear that the default assumptions of linear logic have interesting alternatives. First, the question on when the 'right-to-use' should be specified arises (i.e. either at *definition* or at *use* time, or through a more complex combination of attributes involving both). This is largely problem dependent. A given 'protocol' can be often easily emulated in terms of an alternative, but for a price (in efficiency and/or simplicity). For instance, in the case of distributed languages, allowing *multiple readers* is very important in terms of efficiency. This should be expressed easier than with a combination of consume/produce-again operations as it is done in Linear Logic (the more practically-minded Linda framework provides this as a primitive operation).

A formalization of the subset of linear and intuitionistic logic we have implemented and described in the paper, together with its 'translation semantics' would help to clarify some of the remaining ambiguities.

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.

[17] For instance, in the case of `phrase`, Prolog's DCGs would require on-the-fly preprocessing for meta-interpretation.

A larger efficiently executable subset of linear and intuitionistic logic is intended to be gradually added to BinProlog, with emphases on parametric modules, ability to handle open programs and abductive reasoning. Porting to Wild-Life (and its possible fully compiled sucessor) the facilities described in the paper has been started and will continue in the future.

9 Conclusion

We have presented in a united 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 WAM-level description of multi-stream logic grammars (HAGs) which give EDCG functionality without a preprocessor can be easily replicated in any Prolog system. This suggests that the current, preprocessing based DCG grammar implementation may become obsolete and should not be standardized as part of ISO-Prolog. Based on restricted but powerful special cases of implicitly quantified implication (intuitionistic and linear) and their continuation passing instances, we have shown that virtually all programming techniques involving backtrackable state information can be handled elegantly in our framework, with efficient implementation possible for some of its frequently used instances.

References

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. In Jan Maluszyński and Martin Wirsing, editors, *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (Passau, Germany)*, pages 255–274. Springer-Verlag, LNCS 528, August 1991.

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

[AP90] 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.

[Bak92a] H. Baker. Lively linear lisp–'look ma, no garbage!'. *ACM Sigplan Notices*, 27(8):89–98, August 1992.

[Bak92b] H. Baker. NREVERSAL of Fortune–the Thermodynamics of Garbage Collection. In Y. Bekkers and J. Cohen, editors, *Memory Management International Workshop IWMM 92 Proceedings*, number 637 in Lecture Notes in Computer Science. Springer, September 1992.

[BR92] P. Brisset and O. Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992. ftp: //ftp.irisa.fr/local/lande.

[BR93] P. Brisset and O. Ridoux. Continuations in λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 27–43, Budapest, Hungary, 1993. ftp: //ftp.irisa.fr/local/lande.

[BVG95] J. Balsa, Dahl V., and Pereira Lopez J. G. Datalog grammars for abductive syntactic error diagnosis and repair. In *Proceedings NLULP'95*, May 1995.

[CG89] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.

[DBT93] K. De Bosschere and P. Tarau. Blackboard Communication in Logic Programming. In *Proceedings of the PARCO'93 Conference*, Grenoble, France, September 1993.

[DT95] V. Dahl and P. Tarau. Extending Datalog Grammars. May 1995. to appear in proc. of NLDB'95, Paris.

[DTH94] V. Dahl, P. Tarau, and Y. N. Huang. Datalog Grammars. In *Proc. 1994 Joint Conference on Declarative Programming*, pages 268–282, Peniscola, Spain, September 1994.

[DTMP95] V. Dahl, P. Tarau, L. Moreno, and M. Palomar. Treating Coordination with Datalog Grammars. In *Proceedings of the Joint COMPULOGNET/ELSNET/EAGLES Workshop on Computational Logic For Natural Language Processing*, April 1995. to appear.

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

[HM91] J.S. Hodas and D.A. Miller. Logic programming in a fragment of intuitionistic linear logic. In G. Kahn, editor, *Symp. Logic in Computer Science*, pages 32–42, Amsterdam, The Netherlands, 1991.

[Hod92] J. Hodas. Specifing Filler-Gap Dependency Parsers in a Linear-Logic Programming Language. pages 622–636, Cambridge, Massachusetts London, England, 1992. MIT Press.

[Hod93] J.S. Hodas. Logic programming in intuitionistic linear logic. Phd. thesis, University of Pennsylvania, Department of Computer and Information Science, 1993.

[Mil86] D.A. Miller. A theory of modules for logic programming. In *Symp. Logic Programming*, pages 106–115, Salt Lake City, UT, USA, 1986.

[Mil89a] D. A. Miller. Lexical scoping as universal quantification. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 268–283, Cambridge, Massachusetts London, England, 1989. MIT Press.

[Mil89b] D.A. Miller. A logical analysis of modules in logic programming. *J. Logic Programming*, 6(1–2):79–108, 1989.

[Mil91a] D.A. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.

[Mil91b] D.A. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 255–269, Paris, France, 1991. MIT Press.

[MN86] 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.

[MNPS91] D.A. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, (51):125–157, 1991.

[Ous93] John K. Ousterhout. Tcl/Tk, 1993. Program, publicly available on INTERNET (harbor.ecn.purdue.edu).

[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*.

[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.

[TD94] Paul Tarau and Veronica Dahl. Logic Programming and Logic Grammars with First-order Continuations. In *Proceedings of LOPSTR'94*, Pisa, June 1994.

[TDF95] Paul Tarau, Veronica Dahl, and Andrew Fall. 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*.

[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.

[Wad93] Philip Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, pages 1–17, 1993.

Appendix I: WAM code for HAGs

The reader not interested in low-level details can skip this Appendix. We believe however, that in an experimental framework, the 'reproducibility' of the results is often facilitated by the presence of actual code.

```
/* Macro which tags value trailed cells so that they can be uniformly
   un-trailed, without requireing if-tests later
*/

#define VTRAIL_IT(V,Val) \
{IF_OVER("value trailing",TR_TOP,TrailStk,bp_halt(14)); \
TR_TOP[0]=(term)(Val); TR_TOP[1]= (void *)(1+(cell)(V)); TR_TOP+=2;}

/* Macro for trailing only once by choice point. It allows builtins
to stamp the heap with a new cell so that it will not
trigger further trailing within the same segment */

#define SMART_VTRAIL_IF(Xref,Xval) \
  if((Xref<SAVED_H) && (NONVAR(Xval) || Xval<(cell)SAVED_H)) \
    VTRAIL_IT(Xref,Xval)

Emulator entries:
......
                  case DCG_CONNECT_1:
                    if(!(H=dcg_connect(H,regs,wam,A))) BFAIL()
                  OUT(X(0));
......
                  case DCG_TELL_1:
                    if(!dcg_tell(regs,wam,A)) BFAIL()
                  BNEXT(1);

                  case DCG_TELLING_1:
                    xref=GETREF(g.connect);
                    xval=INPUT_INT(xref-(term)g.connect);
                  OUT(xval);


Functions:

term dcg_init() {
  g.connect=alloc_var_array(MAXDCG); /* initalizes with new variables */
  SETREF(g.connect,g.connect+1); /* sets current DCG-stream */
  return g.connect;
}

int dcg_tell(regs,wam,A)
  register term regs,*A;
  stack wam;
{ cell xval; long ires; int ok=1;
  xval=X(1);
  if(!INTEGER(xval)) ok=0;
  ires=OUTPUT_INT(xval);
  if(!(ires > 0 && ires < MAXDCG)) ok=0;
```

```
  if(!ok) (void)LOCAL_ERR(xval,"bad argument for dcg_stream");
  else
    {
      xval=GETREF(g.connect);
      SMART_VTRAIL_IF(g.connect,xval);
      SETREF(g.connect,g.connect+ires);
    }
  return ok;
}

term dcg_connect(H,regs,wam,A)
  register term H,regs,*A;
  stack wam;
{
  cell xval,head,tail; term xref,new;
  xref=GETREF(g.connect);
  xval=GETREF(xref);
  SMART_VTRAIL_IF(xref,xval);
  new=H;
  MAKE_LIST();
  NEWVAR(head);
  NEWVAR(tail);
  UNIFAIL(new,xval);
  xref=GETREF(g.connect);
  SETREF(xref,tail);
  X(0)=head;
  return H;
}
```

APPENDIX II. Replicating backtrackable objects

It's relatively simple to implement objects on top of fragments of intuitionistic or linear logic, as has been shown e.g in [Mil86, AP90].

We will describe ones [18] which are unusual in the sense that they are directly usable as predicates for creating new instances which, in turn, inherit this 'replication' ability (among others). They make use of the fact that `assumei/1`'s indefinite scope ensures persistence over the current continuation. Note also that our objects will feature backtrackable state. Replacing *assume* predicates with *assert* or write requests to an external database or the distributed Linda system of BinProlog would give objects with a stronger degree of permanence.

```
% tested in BinProlog 3.36

% interface operations: intended to be visible

class(X):-new(X,class).

part(Body,Part):-new_rel(part,Body,Part).

inst(Body,Part):-new_rel(inst,Body,Part).

state(Body,Part):-new_rel(state,Body,Part).


% implementation: intended to be hidden

% our primitive constructors
relation(class_of).
relation(part_of).
relation(inst_of).
relation(state_of).

% creating a new relation
new_rel(Rel,Body,Part):-
  symcat(Body,Part,BodyPart),
  new(BodyPart,class),
  symcat(Rel,of,RelOf),
  Pred=..[RelOf,Body,Part],
  add_rel(Pred).

% creating a new link
new(X,Name):-
  symcat(Name,of,Link), % creates something like '..._of'
  create_new(Link,X,Name). %

create_new(Op,Name,Parent):-
  get_op(Op),
  Rel=..[Op,Name,Parent],
  add_or_call(Name,Op,Rel).

add_or_call(Name,Link,Rel):-
```

[18] To some extent inspired by Tcl/Tk widgets [Ous93].

```
  nonvar(Name),!, % create if Name is given
  add_rel(Rel),
  term_append(Name,arg(X),Pred),
  add_link(Pred,Link,Name,X).
add_or_call(_Name,_Link,Rel):-
  % call otherwise and retrieve _Name and _Link
  Rel.

get_op(Op):-nonvar(Op),!.
get_op(Op):-relation(Op).

add_rel(R):-assumed(R),!,fail.
add_rel(R):-assumei(R).

add_link(Pred,_,_,_):-is_assumed(Pred),!. % assumes at most once
add_link(Pred,Link,Name,X):-
  % this makes Pred have the ability of replication
  assumei( ( Pred:-create_new(Link,X,Name) ) ).

% example of inheritance
is_a(Super,Inst):-assumed(inst_of(Class,Inst)),inherits_from(Super,Class).

inherits_from(X,X).
inherits_from(X,Z):-assumed(class_of(X,Y)),inherits_from(Y,Z).
```

Executing the query:

```
?- class(animal),animal(dog),inst(dog,fido),part(dog,nose),listing.
```

will display the following (hypothetical) world:

```
class_of(animal,class).
class_of(dog,animal).
class_of(dog_fido,class).
class_of(dog_nose,class).

animal(A) :- create_new(class_of,A,animal).
dog(A) :- create_new(class_of,A,dog).
dog_fido(A) :- create_new(class_of,A,dog_fido).

inst_of(dog,fido).
part_of(dog,nose).
dog_nose(A) :- create_new(class_of,A,dog_nose).
```

Note that the replication ability is propagated by passing the `create_new` predicate to newly created objects. By replacing assumei/1 in `add_link/4` with a linear assumption, this ability can be limited. To be able to walk through arbitrary inheritance graphs without looping one can simply replace intuitionistic `assumei/1` in `add_rel/1` with linear `assumel/1`.