Logic Programming and Logic Grammars with Binarization and First-order Continuations

Paul Tarau[1] and Veronica Dahl[2]

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

Abstract. Continuation passing binarization and specialization of the WAM to binary logic programs have been proven practical implementation techniques in the BinProlog system. In this paper we investigate the additional benefits of having first order continuations at source level. We devise a convenient way to manipulate them by introducing multiple-headed clauses which give direct access to continuations at source-level. We discuss the connections with various logic grammars, give examples of typical problem solving tasks and show how looking at the future of computation can improve expressiveness and describe complex control mechanisms without leaving the framework of binary definite programs.
Keywords: continuation passing binary logic programs, logic grammars, program transformation based compilation, continuations as first order objects, logic programming with continuations.

1 Introduction

From its very inception, logic programming has cross-fertilized with computational linguistics in very productive ways. Indeed, logic programming itself grew from the automatic deduction needs of a question-answering system in French [7]. Over the years we have seen other interesting instances of this close-relatedness. The idea of continuations, developed in the field of denotational semantics [22] and functional programming [27, 26] has found its way into programming applications, and has in particular been useful recently in logic programming.

In this article we continue this tradition by adapting to logic programming with continuations some of the techniques that were developed in logic grammars for computational linguistic applications. In particular, logic grammars have been augmented by allowing extra symbols and meta-symbols in the left hand sides of rules [6, 17, 9]. Such extensions allow for straightforward expressions of contextual information, in terms of which many interesting linguistic phenomena have been described. We show that such techniques can be efficiently transfered to continuation passing binary programs, that their addition

motivates an interesting style of programming for applications other than linguistic ones, and that introducing continuations in logic grammars with multiple left-hand-side symbols motivates in turn novel styles of bottom-up and mixed parsing, while increasing efficiency.

2 Motivation

Several attempts to enhance the expressiveness of logic programming have been made over the years. These efforts range from simply providing handy notational variants, as in DCGs, to implementing sophisticated new frameworks, such as HiLog [5, 29].

In λProlog for instance [14, 15] there is a nice facility, that of scoping of clauses, which is provided by the λProlog incarnation of the intuitionistic rule for implication introduction:

$$\boxed{\frac{\Sigma; \Gamma, P \vdash C}{\Sigma; P \vdash \Gamma \Rightarrow C} \Rightarrow_R}$$

Example 1 *For instance, in the λProlog program* [3]*:*

```
insert X Xs Ys :-
    paste X => ins Xs Ys.

ins Ys [X|Ys] :- paste X.
ins [Y|Ys] [Y|Zs]:- ins Ys Zs.
```

used to nondeterministically insert an element in a list, the unit clause `paste X` *is available only within the scope of the derivation for* `ins`*.*

With respect to the corresponding Prolog program we are working with a simpler formulation in which the element to be inserted does not have to percolate as dead weight throughout each step of the computation, only to be used in the very last step. We instead clearly isolate it in a global-value manner, within a unit clause which will only be consulted when needed, and which will disappear afterwards.

Now, let us imagine we are given the ability to write part of a proof state context, i.e., to indicate in a rule's left-hand side not only the predicate which should match a goal atom to be replaced by the rule's body, but also which other goal atom(s) should surround the targeted one in order for the rule to be applicable.

Example 2 *Given this, we could write a program for insert which strikingly resembles the λProlog program given above:*

[3] where for instance `insert X Xs Ys` means `insert(X,Xs,Ys)` in curried notation

```
insert(X,Xs,Ys):-ins(Xs,Ys),paste(X).

ins(Ys,[X|Ys]),paste(X).
ins([Y|Ys],[Y|Zs]):-ins(Ys,Zs).
```

Note that the element to be inserted is not passed to the recursive clause of the predicate **ins/2** (which becomes therefore simpler), while the unit clause of the predicate **ins/2** will communicate directly with **insert/3** which will directly 'paste' the appropriate argument in the continuation.

In this formulation, the element to be inserted is first given as right-hand side context of the simpler predicate `ins/2`, and this predicate's first clause consults the context `paste(X)` only when it is time to place it within the output list, i.e. when the fact `ins(Ys,[X|Ys]),paste(X)` is reached.

Thus for this example, we can obtain the expressive power of λProlog without having to resort to an entirely new framework. As we shall see in the next sections, we can simply use any Prolog system combined with a simple transformer to binary programs [24].

3 Multiple Head Clauses in Continuation Passing Binary Programs

We will start by reviewing the program transformation that allows compilation of logic programs towards a simplified WAM specialized for the execution of binary logic programs. We refer the reader to [24] for the original definition of this transformation.

3.1 The binarization transformation

Binary clauses have only one atom in the body (except for some inline 'builtin' operations like arithmetics) and therefore they need no 'return' after a call. A transformation introduced in [24] allows to faithfully represent logic programs with operationally equivalent binary programs.

To keep things simple we will describe our transformations in the case of definite programs. First, we need to modify the well-known description of SLD-resolution (see [11]) to be closer to Prolog's operational semantics. We will follow here the notations of [25].

Let us define the *composition* operator $\oplus$ that combines clauses by unfolding the leftmost body-goal of the first argument.

Definition 1 *Let* $\mathtt{A_0}\mathtt{:-}\mathtt{A_1},\mathtt{A_2},\ldots,\mathtt{A_n}$ *and* $\mathtt{B_0}\mathtt{:-}\mathtt{B_1},\ldots,\mathtt{B_m}$ *be two clauses (suppose* $n > 0, m \geq 0$*). We define*

$$(\mathtt{A_0}\mathtt{:-}\mathtt{A_1},\mathtt{A_2},\ldots,\mathtt{A_n}) \oplus (\mathtt{B_0}\mathtt{:-}\mathtt{B_1},\ldots,\mathtt{B_m}) = (\mathtt{A_0}\mathtt{:-}\mathtt{B_1},\ldots,\mathtt{B_m},\mathtt{A_2},\ldots,\mathtt{A_n})\theta$$

with $\theta = mgu(\mathtt{A_1},\mathtt{B_0})$*. If the atoms* $\mathtt{A_1}$ *and* $\mathtt{B_0}$ *do not unify, the result of the composition is denoted as* $\perp$*. Furthermore, as usual, we consider* $\mathtt{A_0}\mathtt{:-true},\mathtt{A_2},\ldots,\mathtt{A_n}$

to be equivalent to `A0:-A2,...,An`*, and for any clause* `C`*,* $\bot \oplus$ `C = C` $\oplus \bot$ `=` $\bot$*. We assume that at least one operand has been renamed to a variant with fresh variables.*

Let us call this Prolog-like inference rule LF-SLD resolution (LF for 'left-first'). Remark that by working on the program $P^{'}$ obtained from P by replacing each clause with the set of clauses obtained by all possible permutations of atoms occurring in the clause's body every SLD-derivation on P can be mapped to an LF-SLD derivation on $P^{'}$.

Before defining the binarization transformation, we describe two auxiliary transformations.

The first transformation converts facts into rules by giving them the atom `true` as body. E.g., the fact `p` is transformed into the rule `p :- true`.

The second transformation, inspired by [28], eliminates the metavariables by wrapping them in a `call/1` goal. E.g., the rule `and(X,Y):-X, Y` is transformed into `and(X,Y) :- call(X), call(Y).`

The transformation of [24] (*binarization*) adds continuations as extra arguments of atoms in a way that preserves also first argument indexing.

Definition 2 *Let P be a definite program and $Cont$ a new variable. Let T and $E = p(T_1, ..., T_n)$ be two expressions.*[4] *We denote by $\psi(E, T)$ the expression $p(T_1, ..., T_n, T)$. Starting with the clause*

`(C)` $A : -B_1, B_2, ..., B_n.$

we construct the clause

`(C')` $\psi(A, Cont) : -\psi(B_1, \psi(B_2, ..., \psi(B_n, Cont)))$.

The set P' of all clauses `C'` *obtained from the clauses of P is called the binarization of P.*

Example 3 *The following example shows the result of this transformation on the well-known 'naive reverse' program:*

```
app([],Ys,Ys,Cont):-true(Cont).
app([A|Xs],Ys,[A|Zs],Cont):-app(Xs,Ys,Zs,Cont).

nrev([],[],Cont):-true(Cont).
nrev([X|Xs],Zs,Cont):-nrev(Xs,Ys,app(Ys,[X],Zs,Cont)).
```

These transformations preserve a strong operational equivalence with the original program with respect to the LF-SLD resolution rule which is *reified* in the syntactical structure of the resulting program.

Theorem 1 *([25]) Each resolution step of an LF-SLD derivation on a definite program P can be mapped to an SLD-resolution step of the binarized program P'. Let G be an atomic goal and $G' = \psi(G, true)$. Then, computed answers obtained querying P with G are the same as those obtained by querying P' with G'.*

[4] Atom or term.

Notice that the equivalence between the binary version and the original program can also be explained in terms of fold/unfold transformations as suggested by [18].

Clearly, continuations become explicit in the binary version of the program. We will devise a technique to access and manipulate them in an intuitive way, by modifying BinProlog's binarization preprocessor.

3.2 Modifying the binarization preprocessor for multi-head clauses

The main difficulty comes from the fact that trying to directly access the continuation from 'inside the continuation' creates a cyclic term. We have overcome this in the past by copying the continuation in BinProlog's blackboard (a permanent data area) but this operation was very expensive.

The basic idea of the approach in this paper is inspired by 'pushback lists' originally present in [6] and other techniques used in logic grammars to simulate movement of constituents [9]. We will allow a *multiple head notation* as in:

```
a,b,c:-d,e.
```

intended to give, via binarization:

```
a(b(c(A))) :- d(e(A))
```

This suggests the following:

Definition 3 *A multiheaded definite clause is a clause of the form:*

$$A_1, A_2, \ldots, A_m :- B_1, B_2, \ldots, B_n.$$

A multiheaded definite program is a set of multiheaded definite clauses.

Logically speaking, **','/2** in

$$A_1, A_2, ..., A_m :- B_1, B_2, ..., B_n$$

is interpreted as conjunction on both sides.

The reader familiar with grammar theory will notice (as suggested by [13]) that generalizing from definite to multi-headed definite programs is similar to going from context-free grammars to context-dependent grammars.

The binarization of the head will be extended in a way similar to that of the binarization of the right side of a clause.

Definition 4 *Let P be a multiheaded definite program and $Cont$ a new variable. Starting with the multiheaded clause*

(C) $A_1, A_2..., A_m :- B_1, B_2, ..., B_n.$

we construct the clause

(C') $\psi(A_1, \psi(A_2, ..., \psi(A_m, Cont))) :- \psi(B_1, \psi(B_2, ..., \psi(B_n, Cont))).$

Note that after this transformation the binarized head will be able to match the initial segment of the current implicit goal stack of BinProlog embedded in the continuation, i.e. to look into the immediate future of the computation.

Somewhat more difficult is to implement 'meta-variables' in the left side. In the case of a goal like:

```
a,Next:-write(Next),nl,b(Next).
```

we need a more complex binary form:

```
a(A) :- strip_cont(A,B,C,write(B,nl(b(B,C)))).
```

where `strip_cont(GoalAndCont,Goal,Cont)` is needed to undo the binarization and give the illusion of getting the goal **Next** at source level by unification with a metavariable on the left side of the clause **a/1**.

Notice that the semantics of the continuation manipulation predicate `strip_cont/3` is essentially first order, as we can think of `strip_cont` simply as being defined by a set of clauses like:

```
.........
strip_cont(f(X1,...,Xn,C),f(X1,...,Xn),C).
.........
```

for every functor `f` occuring in the program.

The full code of the preprocessor that handles multiples-headed clauses is given in **Appendix A**.

4 Programming with Continuations in Multiple-headed clauses

The first question that comes to mind is how well our transformation technique performs compared with handwritten programs.

Example 4 *The following program is a continuation based version of* **nrev/2**.

```
app(Xs,Ys,Zs):-app_args(Xs,Zs),paste(Ys).

app_args([],[X]),paste(X).
app_args([A|Xs],[A|Zs]):-app_args(Xs,Zs).

nrev([],[]).
nrev([X|Xs],R):-
  nrev1(Xs,R),
  paste(X).

nrev1(Xs,R):-
  nrev(Xs,T),
  app_args(T,R).
```

which shows no loss in speed compared to the original program (530 KLIPS with BinProlog 2.20 on a Sparcstation 10-40).

One of Miller's motivating examples for intuitionistic implication in λProlog [5] [14], is the reverse predicate with intuitionistic implication.

Example 5 *By using our multi-headed clauses 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]).
```

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).
```

Notice that such definitions are 'virtual' (i.e. supplied by generic WAM-level operations) in BinProlog, for space and efficiency reasons [23].

By replacing `true(C):-C` an appropriate set of clauses where `C=p(X1,...,Xn)` for every head of clause `p/n` occurring in the program, we obtain a definite binary program which accurately describes the operational semantics of the original multi-headed program.

Although the existence of such a translation is expected (as binary definite programs are Turing-equivalent) its simple syntactic nature rehabilitates the idea of using a *translation semantics* as an 'internal' means to describe the semantics of multi-headed logic programs, despite the fact that it fixes *an implementation* as the *meaning* of a programming construct.

The availability of such a programming style in the Horn subset of Prolog is also of practical importance as classical Prolog is still about 5-10 times faster than, for instance, the fastest known λProlog implementation [4].

Example 6 *The following program implements a* **map** *predicate with a Hilog-style (see [5]) syntax:*

[5] which is, by the way, a motivating example also for Andreoli and Pareschi's Linear Objects, [2]

```
cmap(F),i([]),o([]).
cmap(F),i([X|Xs]),o([Y|Ys]):-G=..[F,X,Y],G,cmap(F),i(Xs),o(Ys).

inc10(X,Y):-Y is X+10.

test:-cmap(inc10),i([1,2,3,4,5,6]),o(Xs),write(Xs),nl.
```

Some interesting optimization opportunities by program transformation can be obtained by using multiple-headed clauses, as can be seen in the following example of elimination of existential variables (this technique has been pioneered by Proietti and Pettorossi, in a fold/unfold based framework, [19]).

Example 7 *Given the program:*

```
r(X,Y):-p(X,N),q(N,Y).

p(a,1).    q(1,10).
p(b,2).    q(2,11).
```

we can rewrite it as:

```
r(X,Y):-p(X),result(Y).

p(a) :-q(1).
p(b) :- q(2).

q(1), result(10).
q(2), result(11).
```

which avoid passing unnecessary information by directly unifying the 2 occurrences of `result(_)`.

Example 8 *Multi-headed clauses can be used to ensure direct communication with the leafs of a tree (represented simply as a Prolog term).*

```
frontier(T,X):-leaf_var(T),result(X).

leaf_var(X), result(X):- var(X).
leaf_var(T) :- compound(T), T=..[_|Xs], member(X,Xs), leaf_var(X).
```

where `?-frontier(X)` *returns non-deterministically all the variables in a term seen as a (finite) tree.*

The example also shows that the technique scales easily to arbitrary recursive data-types, and to programs in practice beyond the domain of definite programs.

As it can be seen from the previous examples, the advantage of our technique is that the programmer can follow an intuitive, grammar-like semantics when dealing with continuations and that 'predictability' of the fact that the continuation will be at the right place at the right time is straightforward.

The next section will show some special cases where full access to continuations is more convenient.

5 Multiple headed clauses vs. full-blown continuations

BinProlog 2.20 supports direct manipulation of binary clauses denoted

```
Head ::- Body.
```

They give full power to the knowledgeable programmer on the future of the computation. Note that such a facility is not available in conventional WAM-based systems where continuations are not first-order objects.

Example 9 *We can use them to write programs like:*

```
member_cont(X,Cont)::-
  strip_cont(Cont,X,NewCont,true(NewCont)).
member_cont(X,Cont)::-
  strip_cont(Cont,_,NewCont,member_cont(X,NewCont)).

test(X):-member_cont(X),a,b,c.
```

A query like

```
?-test(X).
```

will return `X=a; X=b; X=c;` `X=`*whatever follows from the calling point of* `test(X)`.

```
catch(Goal,Name,Cont)::-
   lval(catch_throw,Name,Cont,call(Goal,Cont)).

throw(Name,_)::-
   lval(catch_throw,Name,Cont,nonvar(Cont,Cont)).
```

where `lval(K1,K2,Val)` *is a BinProlog primitive which unifies* `Val` *with a backtrackable global logical variable accessed by hashing on two (constant or variable) keys* `K1,K2`.

Example 10 *This allows for instance to avoid execution of the infinite* `loop` *from inside the predicate* `b/1`.

```
loop:-loop.

c(X):-b(X),loop.

b(hello):-throw(here).
b(bye).

go:-catch(c(X),here),write(X),nl.
```

Notice that due to our translation semantics this program still has a first order reading and that BinProlog's `lval/3` is not essential as it can be emulated by an extra argument passed to all predicates.

Although implementation of `catch` and `throw` requires full-blown continuations, we can see that at user level, the multi-headed clause notation is enough.

6 Continuation Grammars

By allowing us to see the right context of a given grammar symbol, continuations can make logic grammars both more efficient and simpler, as well as provide several sophisticated capabilities within a simple framework. We next discuss some of these capabilities.

6.1 Parsing strategy versatility

Changing parsing strategies is usually deemed to require a different parser. We now discuss how we can effect these changes with no more apparatus than continuation grammars.

Take for instance the noun phrase "Peter, Paul and Mary". We can first use a tokenizer which transforms it into the sequence

Example 11 `(noun('Peter'), comma, noun('Paul'), and, noun('Mary'))`

and then use this sequence to define the rule for list-of-names in the following continuation grammar:

```
names  --> noun('Peter'), comma, noun('Paul'), and, noun('Mary').

noun(X), comma --> [X,','].
noun(X), and, noun(Last) --> [X,'and',Last].
```

This is an interesting way in which to implement bottom-up parsing of recursive structures, since no recursive rules are needed, yet the parser will work on lists of names of any length. It is moreover quite efficient, there being no backtracking. It can also be used in a mixed strategy, for instance we could add the rules:

```
verb_phrase --> verb.
verb --> [sing].
```

and postulate the existence of a verb phrase following the list of nouns, which would then be parsed top-down. The modified rules follow, in which we add a marker to record that we have just parsed a noun phrase, and then we look for a verb phrase that follows it.

```
noun(X), comma --> [X,','].
noun(X), and, noun(Last) --> [X,'and',Last], parsed_noun_phrase.

parsed_noun_phrase --> verb_phrase.
```

DCGs allow further left-hand side symbols provided they are terminal ones (it has been shown that rules with non-leading non-terminal left hand side symbols can be replaced by a set of equivalent, conforming rules [6]). However, because their implementation is based on synthesizing those extra symbols into

the strings being manipulated, lookahead is simulated by creating the expectation of finding a given symbol in the future, rather than by actually finding it right away.

With continuations we can implement a more restricted but more efficient version of multiple-left-hand-side symbol-accepting DCGs, which we shall call *continuation grammars*. It is more restricted in the sense that the context that an initial left-hand side symbol is allowed to look at is strictly its right-hand side sisters, rather than any descendants of them, but by using this immediate context right away the amount of backtracking is reduced for many interesting kinds of problems.

Example 12 *For comparison with the length of code, here is a non-continuation based DCG formulation of the "list of names" example which allows for multiple left-hand side symbols* [6]

```
names --> start, start(C), name, end(C), nms, end.

nms --> [].
nms --> start(C), name, end(C), nms.
```

The next two rules are for terminalizing symbols that appear in later rules as non-leading left-hand-side symbols.

```
end --> [end].
start(C) --> [start(C)].

start, [end] --> []
start, [start(C)] --> [].

end(and), end --> [,].
end(comma), start(and) --> [and].
end(comma), start(comma) --> [,].
```

Other possible applications are to procedures to read a word or a sentence, which typically make explicit use of a lookahead character or word, to restrictive relative clauses, whose end could be detected by looking ahead for the comma that ends them, etc.

6.2 Relating of long-distance constituents

One important application in computational linguistics is that of describing movement of constituents. For instance, a common linguistic analysis would state that the object noun phrase in "Jack built the house" moves to the front through relativization, as in "the house that Jack built", and leaves a trace behind in the phrase structure representing the sentence. For describing movement,

[6] This is a simplified version of the corresponding code fragment in [8].

we introduce the possibility of writing one meta-variable in the left hand side of continuation grammar rules. Then our example can be handled through a rule such as the following:

```
relative_pronoun, X, trace --> [that], X.
```

This rule accounts, for instance, for "the house that jack built", "the house that the Prime Minister built", "the house that the man with the yellow hat who adopted curious George built", and so on. A simple grammar containing a similar rule follows, the reader might try it to derive: "The elephant that Jill photographed smiles":

Example 13 `sentence --> noun_phrase,verb_phrase.`

```
noun_phrase --> proper_name.
noun_phrase --> det, noun, relative.
noun_phrase --> trace.

verb_phrase --> verb, noun_phrase.
verb_phrase --> verb.

relative --> [].
relative --> relative_marker, sentence.

relative_marker, X, trace --> relative_pronoun, X.

det --> [the].

noun --> [elephant].

relative_pronoun --> [that].

proper_name --> ['Jill'].

verb --> [photographed].
verb --> [smiles].
```

What is noteworthy about this example is that it shows how to use plain Prolog plus binarization to achieve the same expressive power which used to require more sophisticated grammar formalisms, such as Extraposition or Discontinuous Grammars [17, 9].

6.3 Computation viewed as parsing

Continuation Grammars are useful not only for linguistic or intrinsically grammatical examples, but can also serve to simplify the description of many problems which can be formulated in terms of a grammar.

Example 14 *For instance, we can sort a list of numbers by viewing it as an input string to a grammar, which obtains successive intermediate strings in a parallel-like fashion until two successively obtained strings are the same:*

```
sort(Input):- s(V,V,Input,[]), remember(Input).

s(H,[Y,X|Z]) --> [X,Y], {X>Y}, !, s(H,Z).
s(H,[X|Z]) --> [X], s(H,Z).
s(NewString,[]), {remember(OldString)} -->
  {decide(OldString,NewString)}.

decide(Result,Result),output(Result).
decide(OldString,NewString):- sort(NewString).

par_sort(Input,Result):-sort(Input),output(Result).
```

The first clause initializes a difference-list as empty (both the head and tail are represented by a variable V); adds the two extra arguments (input string and output string) that are typically needed by the Prolog translation of the grammar rules; and pastes a reminder of the last string obtained (initially, the input string).

The first grammar rule consumes the next two numbers from the input string if they are unordered, and calls s recursively, keeping track of their ordered version in the difference list represented by the two first arguments of s.

If the next two numbers to be consumed are not ordered, or if there is only one number left, the second grammar rule consumes one number, and keeps track of it in the difference list as it recursively calls s.

The third grammar rule finds no more numbers to consume, so it consults the last string remembered in order to decide whether to print the result (if the new string is no different than the last one) or to start another iteration of the algorithm on the list last obtained.

The last clause returns the answer directly from the continuation. For instance, on the input string `L=[8,4,5,3]` with the goal `par_sort(L,R)`, we successively obtain: `[4,8,3,5]`, `[4,3,8,5]` and finally `R=[3,4,5,8]`.

The above example is a variation of the odd-even-transposition parallel algorithm for sorting numbers [1]. A grammatical version of this algorithm has been developed in terms of parametric L-systems [20], an extension of L-systems which operates on parametric words and which can be viewed as a model of parallel computation. It is interesting to note that by using continuation grammars we need no special formalisms to obtain the kind of parallel computation that is provided by more elaborate formalisms such as parametric L-systems.

7 Conclusion

We have proposed the use of continuation-passing binarization both for extending logic programs to allow multiple heads, and for a fresh view of those logic

grammars which allow multiple left-hand-side symbols. In both of these areas, the use of continuations has invited interesting new programming (grammar) composition techniques which we have provided examples for.

Other logic programming proposals involve multiple heads, e.g. disjunctive logic programming [12, 21] or contextual logic programming [16, 10]. However, in these approaches the notion of alternative conclusions is paramount, whereas in our approach we instead stress the notion of contiguity that computational linguistics work has inspired. A formal characterization of this stress with respect to logic grammars has been given in [3].

The technique has been included as a standard feature of the BinProlog 2.20 distribution available by ftp from clement.info.umoncton.ca.

Acknowledgment

This research was supported by NSERC Operating grant 31-611024, and by an NSERC Infrastructure and Equipment grant given to the Logic and Functional Programming Laboratory at SFU, in whose facilities this work was developed. We are also grateful to the Centre for Systems Science, LCCR and the Department of Computing Sciences at Simon Fraser University for the use of their facilities. Paul Tarau thanks also for support from the Canadian National Science and Engineering Research Council (Operating grant OGP0107411) and a grant from the FESR of the Université de Moncton.

References

1. S. Akl. *The design and analysis of parallel algorithms.* Prentice Hall, Englewood Cliffs, 1989.
2. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, Jerusalem, Israel, 1990. MIT Press.
3. J. Andrews, V. Dahl, and F. Popowich. A Relevance Logic Characterization of Static Discontinuity Grammars. Technical report, CSS/LCCR TR 91-12, Simon Fraser University, 1991.
4. P. Brisset. Compilation de λProlog. Thèse, Université de Rennes I, 1992.
5. W. Chen and D. S. Warren. Compilation of predicate abstractions in higher-order logic programming. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, number 528 in Lecture Notes in Computer Science, pages 287–298. Springer Verlag, Aug. 1991.
6. A. Colmerauer. *Metamorphosis Grammars*, volume 63, pages 133–189. Springer-Verlag, 1978.
7. A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un systeme de communication homme-machine en francais. Technical report, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, Marseille, 1973.
8. V. Dahl. Translating spanish into logic through logic. *American Journal of Computational Linguistics*, 13:149–164, 1981.

9. V. Dahl. Discontinuous grammars. *Computational Intelligence*, 5(4):161–179, 1989.
10. J.-M. Jacquet and L. Monteiro. Comparative semantics for a parallel contextual logic programming language. In S. Debray and M. Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 195–214, Cambridge, Massachusetts London, England, 1990. MIT Press.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
12. J. Lobo, J. Minker, and A. Rajasekar. Extending the semantics of logic programs to disjunctive logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 255–267, Cambridge, Massachusetts London, England, 1989. MIT Press.
13. J. Maluszyński. On the relationship between context-dependent grammars and multi-headed clauses., June 1994. Personal Communication.
14. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.
15. D. A. Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 268–283, Cambridge, Massachusetts London, England, 1989. MIT Press.
16. L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 284–299, Cambridge, Massachusetts London, England, 1989. MIT Press.
17. F. Pereira. Extraposition grammars. *American Journal for Computational Linguistics*, 7:243–256, 1981.
18. M. Proietti. On the definition of binarization in terms of fold/unfold., June 1994. Personal Communication.
19. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, number 528 in Lecture Notes in Computer Science, pages 347–358. Springer Verlag, Aug. 1991.
20. P. Prusinkiewicz and J. Hanan. *L-systems: from formalism to programming languages*. Springer-Verlag, 1992.
21. D. W. Reed, D. W. Loveland, and B. T. Smith. An alternative characterization of disjunctive logic programs. In V. Saraswat and K. Ueda, editors, *Logic Programming Proceedings of the 1991 International Symposium*, pages 54–70, Cambridge, Massachusetts London, England, 1991. MIT Press.
22. J. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA. The MIT Press, 1977.
23. P. Tarau. Program Transformations and WAM-support for the Compilation of Definite Metaprograms. In A. Voronkov, editor, *Logic Programming, RCLP Proceedings*, number 592 in Lecture Notes in Artificial Intelligence, pages 462–473, Berlin, Heidelberg, 1992. Springer-Verlag.
24. P. Tarau and M. Boyer. Elementary Logic Programs. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173. Springer, Aug. 1990.
25. P. Tarau and K. De Bosschere. Memoing with Abstract Answers and Delphi Lemmas. In Y. Deville, editor, *Logic Program Synthesis and Transformation*, Springer-

Verlag, Workshops in Computing, Louvain-la-Neuve, July 1993.
26. P. Wadler. Monads and composable continuations. *Lisp and Symbolic Computation*, pages 1–17, 1993.
27. M. Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, 1980.
28. D. H. D. Warren. Higher-order extensions to Prolog – are they needed? In D. Michie, J. Hayes, and Y. H. Pao, editors, *Machine Intelligence 10*. Ellis Horwood, 1981.
29. D. S. Warren. The XOLDT System. Technical report, SUNY Stony Brook, electronic document: ftp sbcs.sunysb.edu, 1992.

Appendix A

```
% converts a multiple-head clause to its binary equivalent
def_to_mbin((H:-B),M):-!,def_to_mbin0(H,B,M).
def_to_mbin(H,M):-def_to_mbin0(H,true,M).

def_to_mbin0((H,Upper),B,(HC:-BC)) :- nonvar(H),!,
  termcat(H,ContH,HC),
  add_upper_cont(B,Upper,ContH,BC).
def_to_mbin0(H,B,(HC:-BC)) :- !,
  termcat(H,Cont,HC),
  add_cont(B,Cont,BC).

add_upper_cont(B,Upper,ContH,BC):-nonvar(Upper),!,
  add_cont(Upper,ContU,ContH),
  add_cont(B,ContU,BC).
add_upper_cont(B,Upper,ContH,BC):-
  add_cont((strip_cont(ContH,Upper,ContU),B),ContU,BC).

% adds a continuation to a term

add_cont((true,Gs),C,GC):-!,add_cont(Gs,C,GC).
add_cont((fail,_),C,fail(C)):-!.
add_cont((G,Gs1),C,GC):-!,
  add_cont(Gs1,C,Gs2),
  termcat(G,Gs2,GC).
add_cont(G,C,GC):-termcat(G,C,GC).

strip_cont(TC,T,C):-TC=..LC,append(L,[C],LC),!,T=..L.
```

The predicate termcat(Term,Cont,TermAndCont) is a BinProlog builtin which works as if defined by the following clause:

```
termcat(T,C,TC):-T=..LT,append(LT,[C],LTC),!,TC=..LTC.
```

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