Logic Programming and Logic Grammars with First-order Continuations

Paul Tarau
Département d'Informatique
Université de Moncton
and Logic Programming Group
Simon Fraser University
Email: tarau@cs.sfu.ca

Veronica Dahl
Logic Programming Group
Faculty of Computing Science
Simon Fraser University
Email: veronica@cs.sfu.ca

- Extended Abstract -

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 modifying BinProlog's preprocessor to allow multiple-head 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 control mechanisms in a declarative framework.

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 [4]. Over the years we have seen other interesting instances of this close-relatedness. The idea of continuations, developed in the field of denotational semantics [11] and functional programming [14] 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

[3, 6]. 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 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 [12] for the original definition of this transformation.

2.1 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 [12] 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 [8]) to be closer to Prolog's operational semantics. We will follow here the notations of [13].

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

Definition 1 *Let* $\mathtt{A}_0\mathtt{:-A}_1\mathtt{,A}_2\mathtt{,\ldots,A}_n$ *and* $\mathtt{B}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m$ *be two clauses* ($n > 0, m \geq 0$)*. We define*

$$(\mathtt{A}_0\mathtt{:-A}_1\mathtt{,A}_2\mathtt{,\ldots,A}_n) \oplus (\mathtt{B}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m) = (\mathtt{A}_0\mathtt{:-B}_1\mathtt{,\ldots,B}_m\mathtt{,A}_2\mathtt{,\ldots,A}_n)\theta$$

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* $\mathtt{A}_0\mathtt{:-true,A}_2\mathtt{,\ldots,A}_n$ *to be equivalent to* $\mathtt{A}_0\mathtt{:-A}_2\mathtt{,\ldots,A}_n$*, and for any clause* $\mathtt{C}$*,* $\perp \oplus \mathtt{C} = \mathtt{C} \oplus \perp = \perp$*. We assume (when necessary) 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 [15], 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 [12] (*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.*[1] *We denote by $\psi(E,T)$ the expression $p(T_1, ..., T_n, T)$. Starting with the clause*

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

we construct the clause

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

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

Example 1 *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 *(Tarau and De Bosschere, [13]) 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'.*

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.

[1] Atom or term.

2.2 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 overcomed 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 [3] and other techniques used in logic grammars to simulate movement of constituents [6].

First, we will allow a *multiple head notation* as in:

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

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

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

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**.

```
% 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.
```

3 Programming with Continuations in Multiple-headed clauses

We will give some examples of programs. The main advantage of our technique is that the programmer can follow an intuitive, grammar-like semantics when dealing with continuations.

Example 2 *The following program inserts an element in a list:*

```
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** (which becomes therefore simpler), while the unit clause of the predicate **ins** will communicate directly with **insert** which will directly 'paste' the appropriate argument in the continuation.

Example 3 *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 show no loss in speed compared to the original program (530 KLIPS on a Sparcstation 10-40).

Example 4 *The following program implements a* **map** *predicate with a Hilog-style (see [2]) syntax:*

```
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.
```

An interesting further development we intend to describe in the full paper is to investigate the optimization opportunities by program transformation using multiple-headed BinProlog clauses.

4 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.

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).

In the full paper we will compare the full-blown continuation passing binary programs with their multi-head counterparts.

5 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. For instance, for describing a noun phrase such as "Peter, Paul and Mary", we can first use a tokenizer which transforms it into the sequence

```
(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 (the rule will have an arbitrary name such as "test", since the tokenizer will not know how to give it a mnemonic name):

```
test  --> 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 [3]). 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.

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 [2]

```
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.

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 gap behind in the phrase

[2]This is a simplified version of the corresponding code fragment in [5].

structure representing the sentence. For describing movement, 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, noun_phrase --> [that], X, gap.
```

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. (This example will be expanded in the full paper).

6 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 [1, 10] or contextual logic programming [9, 7]. 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 [1].

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 School 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] 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.

[2] 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.

[3] A. Colmerauer. Metamorphosis grammars. In L. Bolc, editor, *Natural Language Communication with Computers*, volume 63 of *Lecture Notes in Computer Science*, pages 133–189. Springer-Verlag, 1978.

[4] 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.

[5] V. Dahl. Translating Spanish into Logic through Logic. *American Journal of Computational Linguistics*, 13:149–164, 1981.

[6] V. Dahl. Discontinuous grammars. *Computational Intelligence*, 5(4):161–179, 1989.

[7] 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.

[8] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[9] 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.

[10] 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.

[11] J. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* Cambridge, MA. The MIT Press, 1977.

[12] 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.

[13] P. Tarau and K. De Bosschere. Memoing with abstract answers and delphi lemmas. In *Proceedings of the LOPSTR Conference*, Louvain-la-Neuve, Belgium, July 1993. to appear in LNCS.

[14] M. Wand. Continuation-based program transformation strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, 1980.

[15] D. H. 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.