Continuations of logic programs

H.Peter Gumm

Dept. of Mathematics and Computer Science
SUNY College at New Paltz, New Paltz, N.Y.,12561
`gummp@snynewba.bitnet`

0. Background

In the realm of functional programming a wealth of techniques have been explored to transform a program into another equivalent program with the transformed program exhibiting certain computational advantages over the original. Often the transformation involves a "generalization" of the original task, where this generalization requires the addition of further parameters, called "accumulating" parameters. This technique is particularly useful in transforming functional programs into tailrecursive form. A typical example of such a transformation is the generalization of a linearly recursive function such as "factorial" into a tailrecursive function $fact'(n, m) = fact(n) * m$.

On first sight, such generalizations appear to involve quite an insight into the particular problem at hand, but they turn out to be instances of a very general method of transformations based on "continuations". The transformation always succeeds on linearly recursive programs, insight is only requested to further simplify the resulting program.

A *continuation* is a function of one parameter, representing some remaining computation necessary to transform an intermediate value into a final outcome. In the example of the standard definition of the factorial function, after having finished the inner recursive call to $f(n-1)$ the resulting value still has to be multiplied with n, so the continuation would be $\lambda w.n * w$. A representation of this continuation is all that has to be stored in the accumulating parameter. In the preceding case it suffices to simply represent $\lambda w.n*w$ by n. Further optimizations are possible, if the space of representations can be endowed with a monoid structure so the abstraction function maps this monoid homomorphically to the monoid of continuations with composition (denoted by $\circ$) as operation. In the "factorial" example, the monoid is the multiplication monoid on the natural numbers, so $abs(n*m) = \lambda w.(n*m)*w = \lambda w.n*(m*w) = \lambda w.n*w \circ \lambda w.m*w = abs(n) \circ abs(m)$. An excellent account of the technique is given in [W].

The general method of transforming a linearly recursive functional program into tailrecursive form can then be sketched briefly as follows. Let

$$f(x) = \textbf{ if } g(x) \textbf{ then } h(x) \textbf{ else } \Phi(x, f(r(x)))$$

be a linearly recursive program. Generalize it to a function $cf(x,\gamma)$ by introducing a further parameter γ to represent a continuation with the intention

$$cf(x,\gamma) := (\gamma \circ f)(x).$$

Then

$$\begin{aligned} cf(x,\gamma) &= \textbf{ if } g(x) \textbf{ then } (\gamma \circ h)(x) \textbf{ else } \gamma(\Phi(x, f(r(x))) \\ &= \textbf{ if } g(x) \textbf{ then } (\gamma \circ h)(x) \textbf{ else } (\gamma \circ \lambda w.\Phi(x,w) \circ f)r(x) \\ &= \textbf{ if } g(x) \textbf{ then } (\gamma \circ h)(x) \textbf{ else } cf(r(x), \gamma \circ \lambda w.\Phi(x,w)), \end{aligned}$$

a function which is now tailrecursive. The original function f can be recreated using the identity function id in $f(x) = cf(x, id)$.

The second argument to cf, which is a function, can be represented (encoded) by the pieces of data γ and x as $p(x,\gamma)$ with some constructor p, and with a constant id serving as the representation of the identity function. As long as p is a free constructor, i.e. it essentially pushes values on a stack beginning with the empty stack id, we can uniquely decode the function it represents. The decoding map is defined on the space of representations by

$$decode(id) = \lambda x.x$$

and

$$decode(p(x,y)) = decode(y) \circ \lambda w.\Phi(x,w).$$

In many cases simpler representations can be found by means of a binary operation $*$ defined on the space of representations, so that

$$decode(p(x*y, z)) = decode(p(x, p(y,z))),$$

in particular, such a $*$ can always be chosen associative.

1. Logic Programs

In the mathematical semantics of logic programs the order of the predicates in a clause should not matter, but of course it does make a difference to the termination properties of a PROLOG program. More importantly, most practical logic programs contain nonlogical predicates, such as arithmetical predicates, predicates causing side effects or any system predicates that require certain arguments to be bound before execution. Clearly, such predicates cannot be freely permuted with other predicates of the same clause.

Thus the notion of a "tailrecursive" predicate makes sense even in logic programming and indeed most PROLOG compilers will generate more efficient code if a program is tailrecursive. Interpreters will also have to store less backtrackpoints, if a recursive call is the last call in the last applicable clause in the program. Several authors have studied how to apply the unfold/fold technique to transform logic programs into tailrecursive form [TS], [D].

Here we explore a possible way how to give a meaning to continuations in logic programming. There are various ways of doing so , and the continuation may either represent an extra goal to be solved, or a relation between intermediate values and output values, in the case of a logic program whose intended use is to transform input into output. Following this, but independent of the method, simplifications may be applied on the ensuing program, often getting rid of the auxiliary predicate.

A logic program for a predicate `q` is called *linearly recursive*, if there is at most one recursive call to `q` in each clause . `q` is called *tailrecursive*, if it is linearly recursive and each call to `q`, if any, occurs as the last goal in the clauses body. Thus a typical linearly recursive program would be of the form :

$$\mathtt{q}_i(\mathtt{r}_j) \ \ :- \ \mathtt{g}_i(\mathtt{s}_j). \tag{1}$$
$$\mathtt{q}_i(\mathtt{t}_j) \ \ :- \ \mathtt{h}_i(\mathtt{u}_j), \mathtt{q}_i(\mathtt{v}_j), \mathtt{p}_i(\mathtt{w}_j). \tag{2}$$

where we abbreviate termlists such as $\mathtt{t}_1(x_1,\ldots,x_n),\ldots,\mathtt{t}_k(x_1,\ldots,x_n)$ by $\mathtt{t}_i(\mathtt{x}_j)$. There may be at most one recursive call to q in the body of each clause, yet there may be several clauses such as (2).

The following predicate will serve as a prototype to demonstrate the transformation. The formulation for a general linear recursive program will be obvious, but tedious.

The predicate relates a list to its length and can be used either to calculate the length of a list, or to provide a list of a given length. It is obviously linearly recursive, but not tailrecursive, and reordering of the subgoals in its body is not possible, since `V` must be bound, when the predicate "`U is V + 1`" is encountered.

```
length([ ],0).
length([H|T],U)  :- length(T,V),U is V + 1.
```

The idea corresponding to the continuation transformation in functional programming would be to generalize the "length" predicate so that it also incorporates the calculation ensuing after the recursive call in its body, creating a new generalized predicate that will become tailrecursive. For this we extend `length` by a further argument position that is to encode the "ensuing calculation".

In functional programs this ensuing calculation is represented as a function of one argument, the "continuation", in logic programming it ought to be a goal, representing the task yet to be solved. The new predicate, say `cLength`, is intended to have the semantics :

$$\texttt{cLength(L,N,}\Gamma\texttt{)} \Longleftrightarrow \texttt{length(L,N),}\Gamma. \tag{3}$$

The relationship between `length` and `cLength` would then be defined as

$$\texttt{length(L,N)} \;:-\; \texttt{cLength(L,N,true)}. \tag{4}$$

Using (3) as a definition of `cLength` we shall have to remove the reference to the old `length`-predicate. Partial evaluation (see [V],[K]) of `length(L,N)` in (3) yields the clauses :

$$\texttt{cLength([ ],0,}\Gamma\texttt{)} \;:-\; \Gamma. \tag{5}$$

$$\texttt{cLength([H|T],U,}\Gamma\texttt{)} \;:-\; \texttt{length(T,V),U is V+1,}\Gamma. \tag{6}$$

Now, we have to fold the right hand side with (3) which is trivial by simply letting the new continuation be the conjunction of the goals "`U is V+1`" and "Γ", i.e.

$$\texttt{cLength([H|T],U,}\Gamma\texttt{)} \;:-\; \texttt{cLength(T,V,(U is V+1,}\Gamma\texttt{))}. \tag{7}$$

The new program for length consists of (4), (5), and (7) . Some PROLOG implementations would need to replace the call to Γ in the body of (5) with an explicit "`call(Γ)`".

It is clear that the new program for `length` is equivalent to the old version and also that the new program has become tailrecursive. Instead of creating backtrackpoints as is necessary in a call to (6), the extra argument in `cLength` is used to store the necessary information for the ensuing goals. The necessary information to recreate the "ensuing calculation" is completely provided by the variables `U,V` and Γ, so we choose a function symbol `p` to encode the continuation as `p(U,V,Γ)`, with a constant `done` representing the goal `true`. Of course, we need a decoding predicate `run` now, which is given by

```
run(done).
run(p(U,V,G)) :- U is V+1,run(G).
```

The new cLength then becomes :

```
cLength'([ ],0,G)   :- run(G).
cLength'([H|T],U,G) :- cLength'(T,V,p(U,V,G)).
```

with initial call :

$$\texttt{length}(\texttt{L},\texttt{N}) \ :- \ \texttt{cLength}'(\texttt{L},\texttt{N},\texttt{done}).$$

Thus, not surprisingly, forming `p(U,V,G)` amounts to pushing `U` and `V` onto the stack `G`, with `done` representing the empty stack. Clearly, also, there are some savings possible, since not both `U` and `V` need to be pushed onto the stack, in particular, since it is obvious that `V` is local to the body of (6), but we shall see a refined version of the transformation and with it an improved version of `length` in the next section.

2. List recursion.

There may be several reasons why a linear recursive program cannot be turned into a tailrecursive program by simply switching the order of the subgoals in the clauses of (2). In the majority of cases though, there will be some value computed in the recursive call to `q` which is subsequently needed by `p`. (If `q` and `p` do not have any variables in common there is no reason why they could not be interchanged, unless they create sideeffects.) Taking this fact into account, we can improve upon the previous transformation. In this chapter we shall demonstrate this for programs recursing over lists, and in the following chapter we give an example of the same transformation in the context of graphs.

List recursion seems to be a rather typical case where linear recursive programs arise. (Stretching this point somewhat, recursion over natural numbers can be seen as a special case of list recursion). The general form of a program recursing over lists can be written as

$$\texttt{q}([\ ],\texttt{c}). \tag{8}$$

$$\texttt{q}([\texttt{H}|\texttt{T}],\texttt{M}) \ :- \ \texttt{q}(\texttt{T},\texttt{K}),\texttt{r}(\texttt{H},\texttt{K},\texttt{M}). \tag{9}$$

(Additional nonrecursive clauses or additional goals in (8) and (9) would only complicate notation). The body of (9) can be viewed as a relational product (join) of the relations `q(-,-)` and `r(H,-,-)`. Thus continuations should become relations and they should be composed using relational composition $\circ$. Augmenting `q` with a further argument to hold the representation of a continuation and introducing a ternary relation `abs(-,-,-)` that decodes the representation of a continuation, so that $\texttt{abs}(\texttt{rep}(\texttt{C}),-,-) = \texttt{C}$, we introduce the generalization `cq(-,-,-)` of `q(-,-)` with the intention

$$\texttt{cq}(\texttt{L},\texttt{M},\texttt{R}) \Longleftrightarrow \texttt{q}(\texttt{L},\texttt{U}),\texttt{abs}(\texttt{R},\texttt{U},\texttt{M}).$$

To recover the original predicate, we set :

$$\texttt{q}(\texttt{L},\texttt{M}) \ :- \ \texttt{cq}(\texttt{L},\texttt{M},\texttt{id}).$$

and

$$\texttt{abs}(\texttt{id},\texttt{X},\texttt{X}).$$

Next we use the defining clauses for q to partially evaluate the definition of `cq` :

$$\begin{aligned}\mathtt{cq}([\ ],\mathtt{M},\mathtt{R}) &\ :-\ \mathtt{abs}(\mathtt{R},\mathtt{c},\mathtt{M}).\\ \mathtt{cq}([\mathtt{H}|\mathtt{T}],\mathtt{M},\mathtt{R}) &\ :-\ \mathtt{q}(\mathtt{T},\mathtt{K}),\mathtt{r}(\mathtt{H},\mathtt{K},\mathtt{U}),\mathtt{abs}(\mathtt{R},\mathtt{U},\mathtt{M}).\end{aligned}$$

We need the right hand side to be of the form `q(T,K), abs(`Ω`,K,M),` so Ω must encode `H` and `R`. Hence we choose a binary function symbol `p` and the clause

$$\mathtt{abs}(\mathtt{p}(\mathtt{H},\mathtt{R}),\mathtt{K},\mathtt{M})\ :-\ \mathtt{r}(\mathtt{H},\mathtt{K},\mathtt{U}),\mathtt{abs}(\mathtt{R},\mathtt{U},\mathtt{M}).$$

This gives us the final program

$$\mathtt{q}(\mathtt{L},\mathtt{M})\ :-\ \mathtt{cq}(\mathtt{L},\mathtt{M},\mathtt{id}).$$

$$\begin{aligned}\mathtt{cq}([\ ],\mathtt{M},\mathtt{R}) &\ :-\ \mathtt{abs}(\mathtt{R},\mathtt{c},\mathtt{M}).\\ \mathtt{cq}([\mathtt{H}|\mathtt{T}],\mathtt{M},\mathtt{R}) &\ :-\ \mathtt{cq}(\mathtt{T},\mathtt{M},\mathtt{p}(\mathtt{H},\mathtt{R})).\end{aligned}$$

$$\begin{aligned}&\mathtt{abs}(\mathtt{id},\mathtt{X},\mathtt{X}).\\ &\mathtt{abs}(\mathtt{p}(\mathtt{H},\mathtt{R}),\mathtt{K},\mathtt{M})\ :-\ \mathtt{r}(\mathtt{H},\mathtt{K},\mathtt{U}),\mathtt{abs}(\mathtt{R},\mathtt{U},\mathtt{M}).\end{aligned}$$

Here all predicates are tailrecursive. The functor `p` is free, that is, the data structure built as representation of the continuation is isomorphic to a stack, with `id` corresponding to the empty stack. If there were several clauses in the original program containing a call to `q`, we would need a constructor $\mathtt{p}_i$ for each of them, together with a corresponding clause for `abs`.

Suppose that q is called with its first argument bound to a list `l`, then the role of `cq` is merely to push the elements of `l` so they can be retrieved by abs and processed in reverse order.

In special cases various optimizations are possible. If, for example, `r(H,K,M)` does not depend on `H`, such as in the `length` predicate of the previous chapter, then `p` becomes essentially unary and the continuations can be represented by natural numbers, `id, p(id), p(p(id)),..` . Another important case is when a binary operation $\diamond$ can be defined such that $\mathtt{r}(\mathtt{x},-,-)\circ\mathtt{r}(\mathtt{y},-,-) = \mathtt{r}(\mathtt{x}\diamond\mathtt{y},-,-)$. W.l.o.g. we can assume a right unit `e` with `r(e,X,X)`. Then the transformed program simplifies to

$$\mathtt{q}(\mathtt{L},\mathtt{M})\ :-\ \mathtt{cq}(\mathtt{L},\mathtt{M},\mathtt{e}).$$

$$\begin{aligned}\mathtt{cq}([\ ],\mathtt{M},\mathtt{R}) &\ :-\ \mathtt{r}(\mathtt{R},\mathtt{c},\mathtt{M}).\\ \mathtt{cq}([\mathtt{H}|\mathtt{T}],\mathtt{M},\mathtt{R}) &\ :-\ \mathtt{cq}(\mathtt{T},\mathtt{M},\mathtt{H}\diamond\mathtt{R}).\end{aligned}$$

and `abs` becomes superfluous. (Since any call to the program will be made through a call to `q`, the last argument of `cq` will always be bound.) Examples of programs amenable to the latter simplification are e.g. programs combining the elements of a list by an associative operation. The `length` program, again, is a special case here, setting $\mathtt{e} = 0$ and $\mathtt{H}\diamond\mathtt{R} := \mathtt{R} + 1$.

3. Modifying a search program.

The previous transformation is taylored to, but not limited to programs recurring over lists. As an example, suppose a graph is given by a relation `edge(_,_)` relating pairs of nodes. Reachability can then be defined as the transitive hull of the edge relation :

```
reach(X, X).
reach(X, Y)  :- reach(X, Z), edge(Z, Y).
```

The predicate cReach will be introduced again, with the intention:

$$\mathtt{cReach(X,Y,G)} \Longleftrightarrow \mathtt{reach(X,U), abs(G,U,Y).}$$

This leads to

```
reach(X, Y)  :- cReach(X, Y, done).

abs(done, X, X).
```

Partially evaluating the body of this definition we get

```
cReach(X, Y, G)  :- abs(G, X, Y).
cReach(X, Y, G)  :- cReach(X, Y, p(G)).

abs(p(G), Z, Y)  :- edge(Z, U), abs(G, U, Y).
```

The domain of continuation representations, again, is isomorphic to the natural numbers, and it seems that renaming `abs` into `distance` is more appropriate, we get :

```
reach(X, Y)  :- cReach(X, Y, 0).


cReach(X, Y, N)  :- distance(X, Y, N).
cReach(X, Y, N)  :- cReach(X, Y, succ(N1)).

distance(X, X, 0).
distance(X, Y, succ(N))  :- edge(X, U), distance(U, Y, N).
```

Thus, whereas in the original program a call such as `reach(a,b)` results in a depth first search backwards from `b`, the transformed program will do an exhaustive search, increasing the boundaries of the search space with each call to `cReach`. The original program, by contrast, is likely to be caught in infinite loops. Logically, though, the two programs are equivalent.

4. Difference lists.

Difference lists are a representation of the list data structure, that is particularly efficient for the "append" operation, in that appending of two difference lists can be achieved totally by unification(see [ZG]). A difference list `d(A,B)` represents a list that satisfies $\mathtt{d(A,B)} \oplus \mathtt{B} = \mathtt{A}$, where `A` and `B` are lists and $\mathtt{A} \oplus \mathtt{B}$ denotes the list obtained by appending `A` to `B`. The program

```
append(d(X, Y), d(Y, Z), d(X, Z)).
```

appends two difference lists and, obviously, leaves all the work to the unification routine. Difference lists are particularly useful in parsing, where the append program is used to split a list of incoming tokens into pieces. Each piece is then parsed by parsers responsible for the individual nonterminals of the grammar. Applying the continuation transformation onto a simple minded version of a parser we shall see that difference lists quite naturally come about as representations of continuations. Let us take a typical clause of a grammar such as

```
< sentence >::=< nounPhrase >< verbPhrase >< nounPhrase >
```

and a corresponding parser that works on a list of tokens to construct an abstract syntax tree:

```
pSent(In, mkSent(N, V, M))  :- pNP(A, N),
                               append(A, R1, In),
                               pVP(B, V),
                               append(B, R2, R1),
                               pNP(C, M),
                               append(C, [], R2).
```

together with some simple definitions of `pNP` and `pVP` such as e.g.

```
pNP([the, X], subj(the, X))  :- noun(X).

pNP([Y], person(Y))  :- name(Y).

pVP([eats], verb).
pVP([likes], verb).
```

The last call to `append`, in `pSent` could, of course, be dispensed with, but it serves to show the regular structure of the parser. Every subparser is now paired with its own continuation. Since those all have the same structure, we need only one representation, resp. decoding predicate to work for all. This yields:

```
cpNP(N, p(R1, In))  :- pNP(A, N), append(A, R1, In).
cpVP(V, p(R2, R1))  :- pVP(B, V), append(B, R2, R1).
```

```
pSent(In, mkSent(N, V, M))  :- cpNP(N, p(R1, In)),
                               cpVP(V, p(R2, R1)),
                               cpNP(M, p([ ], R2)).
```

partial evaluation of `cpNP` and `cpVP` gives

```
cpNP(subj(the, X), p(R1, In))  :- noun(X), append([the, X], R1, In).
cpNP(person(Y), p(R1, In))  :- name(Y), append([Y], R1, In).

cpVP(verb, p(R2, R1))  :- append([eats], R2, R1).
cpVP(verb, p(R2, R1))  :- append([likes], R2, R1).
```

Next, we can partially evaluate the append subgoals, obtaining:

```
cpNP(subj(the, X), p(R1, [the, X|R1]))  :- noun(X).
cpNP(person(Y), p(R1, [Y|R1]))  :- name(Y).

cpVP(verb, p(R2, [eats|R2])).
cpVP(verb, p(R2, [likes|R2])).
```

Note now, that the previous continuation representation has turned into a difference list, since `p(B,A)` can be interpreted as the difference list `d(A,B)`. Actually, one would probably want to relinquish the constructor `p` altogether, and simply list the components in separate argument positions resulting in the final program, that is a substantial improvement over the initial program, since the calls to `append` have disappeared.

```
pSent(In, mkSent(N, V, M))  :- cpNP(N, R1, In),
                               cpVP(V, R2, R1),
                               cpNP(M, [ ], R2).

cpNP(subj(the, X), R1, [the, X|R1])  :- noun(X).
cpNP(person(Y), R1, [Y|R1])  :- name(Y).

cpVP(verb, R2, [eats|R2]).
cpVP(verb, R2, [likes|R2]).
```

5. Left recursion

Left recursion is a problem frequently encountered in constructions of recursive descent parsers. Its solution is well known, we will nevertheless derive it here again, to show that it may as well be considered an instance of a continuation based transformation. Let the grammar be given as $A \Leftarrow \alpha \mid A\beta$, and let pA, p$\alpha$ and pβ be the associated PROLOG predicates. We disregard as inessential here the fact that pA, pα, pβ usually would have some arguments. The PROLOG program for the grammar,

$$\begin{aligned} \texttt{pA} \ &:- \ \texttt{p}\alpha. \\ \texttt{pA} \ &:- \ \texttt{pA}, \texttt{p}\beta. \end{aligned}$$

would suffer from left recursion. Introducing a continuation parameter and a decoding predicate `abs(_)`, so that

$$\texttt{cpA(G)} \Longleftrightarrow \texttt{pA}, \texttt{abs(G)}$$

we get

$$\begin{aligned} \texttt{pA} \ &:- \ \texttt{cpA(id)}. \\ \texttt{cpA(G)} \ &:- \ \texttt{p}\alpha, \texttt{abs(G)}. \\ \texttt{cpA(G)} \ &:- \ \texttt{cpA(s(G))}. \end{aligned}$$

where

$$\begin{aligned} &\texttt{abs(id)}. \\ &\texttt{abs(s(G))} \ :- \ \texttt{p}\beta, \texttt{abs(G)}. \end{aligned}$$

Once again, the continuations can be represented by the natural numbers, moreover, the last clause for cpA together with the fact that the original call to cpA is with argument `id`, indicate that the argument is really superfluous. We replace it by "_", and it turns out that `abs(_)` will also be called with argument "_", thus we can eliminate the continuation parameter from `cpA` and from `abs`, obtaining the final program

$$\begin{aligned} \texttt{pA} \ &:- \ \texttt{cpA}. \\ \texttt{cpA} \ &:- \ \texttt{p}\alpha, \texttt{abs}. \\ \texttt{abs}&. \\ \texttt{abs} \ &:- \ \texttt{p}\beta, \texttt{abs}. \end{aligned}$$

which is the familiar transformation for leftrecursive grammars.

6. Conclusion

We have demonstrated that the concept of continuation based transformations can be successfully carried over from functional programming to logic programming. Various known techniques of logic programming, such as removal of linear recursion, parsing by difference lists and removal of left recursion in grammars can be considered as special instances of the technique.

7. References

[D] S. Debray "Optimizing Almost-Tail-Recursive Prolog Programs," *Proc. IFIP International Conference on Functional Programming Languages and Computer Architecture.* Nancy, France, 1985.

[K] H. J. Komorowski "Partial evaluation as a means for inferencing data structures in an applicative language : A theory and implementation in the case of Prolog," *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, 255–267 (1982).

[TS] S. Tamaki and T. Sato "Unfold/Fold Transformations of Logic Programs," *Proc. 2nd. Logic Programming Conference*, Uppsala, Sweden, 1984.

[V] R. Venken "A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query optimization," in *T.O'Shea (ed.): ECAI-84. Advances in Artificial Intelligence*, Pisa, Italy, 91–100. North-Holland, 1984.

[W] M. Wand "Continuation based program transformation strategies," *Journal of the ACM,* **27**(1980)164–180.

[ZG] J. Zhang and P. W. Grant "An Automatic Difference-list Transformation Algorithm for Prolog," in *Proceedings of ECAI-88. European Conf. on Artificial Intelligence*, Munich 1988.