A Transformation Based on the Equality between Terms*

Ulrich Neumerkel
Institut für Computersprachen, Technische Universität Wien
ulrich@mips.complang.tuwien.ac.at

Abstract

We present a new transformation of Prolog programs preserving operational equivalence. Our transformation — EBC (equality based continuation) transformation — relies on the introduction of equations between terms. These equations are used to introduce alternative and more efficient representations of terms. When applied to binary Prolog programs, EBC is able to perform the following optimizations by mere source to source transformations: removal of existential variables in programs using difference lists and accumulators, reduction of the number of occur-checks, interprocedural register allocation when executed on the WAM, linearization of recursions, optimization of continuation-like user data structures.

1 Introduction

The limitations of fold/unfold transformations. Currently, most program transformation schemes for Prolog programs and logic programs are based on the framework of fold/unfold transformations as defined by [24] or [16]. This framework is an adaption of fold/unfold transformations originally developed within functional languages [6]. Fold/unfold transformations introduce equalities between logic programs expressed at the level of 'control structures', i.e., predicates, clauses and goals. Transformations within fold/unfold are therefore able to improve or specialize a program on the level of the control structures. They allow a programmer to write more generic and reusable programs and specialize them thereafter, in particular, with the strategy of partial evaluation.

However, the 'data structures' of logic programs, i.e., terms play only a secondary role in these transformations. Fold/unfold transformation do not define directly any transformation rules for terms[1]. By reasoning on the goal level, current fold/unfold transformations are unable to transform a given dynamic data structure in another different dynamic data structure. E.g., a program manipulating lists can only be transformed into a program that either contains no corresponding lists at all or that contains (parts of) those very lists, e.g., in

*To appear in Logic Program Synthesis and Transformation, LOPSTR 1993, Springer-Verlag

[1]Note that the fold/unfold framework is in principle able to describe any transformation desired because the rules for goal replacement (Chapter 3 [24]) allow to replace a goal by any 'equivalent' one. However, no transformations capable of transforming data structures are known in the literature.

the form of a difference-list. Similarly, redundancies between goals cannot be removed in many cases because a goal can only be absorbed or transformed to a different goal. For example, it is impossible to remove all existential variables attached to the difference list in programs corresponding to grammar rules.

Our transformation overcomes the deficiencies of fold/unfold transformations currently in use by using equivalences between terms instead. When our transformation is applied to binary Prolog we are able to improve programs on the goal level as well since goals are encoded as terms.

Binary Prolog. Binary Prolog [25] corresponds to the notion of continuation passing style (CPS) [26] in functional programming languages. When encoding Prolog with the help of binary Prolog using Tarau's transformation [25] Prolog's AND-control is encoded with terms. These new terms are called continuations corresponding to closures in functional programming languages [1]. There are several advantages of using binary Prolog: First, binary Prolog is simpler to implement. Second, binary programs are better amenable to source to source transformations, in particular, our EBC-transformation.

EBC-transformations. In order to change the representation of a program we introduce alternative representations of terms. Conceptually, the syntactic unification of Prolog is extended by new equations that do not alter the behavior of unification. In general we therefore need to introduce new function symbols. The new equations describe alternative and often more efficient representations of terms. Our approach of extending unification is quite different from other approaches like CLP. In EBC we transforme ordinary Prolog programs without any extension while CLP provides extensions to unification visible to the user. We restrict the additional equations to cases that can be implemented with syntactic unification only. In particular we restrict ourselves to terms that serve as continuations. In this case the extended unification can be implemented with syntactic unification.

Contents. We open our presentation with a detailed example in Sect. 2. This example shows how difference lists and in particular grammar rules can profit from our optimization. In Sect. 3 we present the general framework of EBC-transformation underlining the basic notion of *conservative extension* to unification. The optimization of context arguments is discussed in Sect. 4. General strategies for EBC are discussed in Sect. 5. We present all equational schemes developed so far. Finally, related work is found in Sect. 6.

2 Transformation of difference lists

This section shows how EBC treats typical programs with difference lists. Such programs are a source of inefficiency in current Prolog implementations when compared to their procedural counterparts.

Notation. subcontinuations are underlined and **changed parts** are bold. Equations introduced are written $A \doteq B$ to avoid confusion with Prolog's =/2

and $\doteq$ used in unification theory. We do not distinguish between function and predicate symbols in binary programs. Symbols in an equation refer to both.

2.1 A first example

We present an informal derivation in seven steps of a simple program containing an existential variable within a difference list. The formal transformation rules are described in Section 3. The predicate expr/3 describes the relation between a difference list containing terminal symbols and the corresponding abstract syntax tree. Fold/unfold frameworks are unable to remove Xs1[2].

```
expr(t(T)) ⟶                    expr(t(T), [t(T)|Xs],Xs).
    [t(T)].                     expr(node(TL,TR), [op|Xs0],Xs) ←
expr(node(TL,TR)) ⟶                 expr(TL, Xs0,Xs1),
    [op],                           expr(TR, Xs1,Xs).
    expr(TL),
    expr(TR).
```

1st step: binary form. First the program is transformed into binary form with Tarau's transformation [25]. A new argument is added to represent the continuation in an explicit manner. The goals after the first goal in the body are encoded as function symbols placed in the continuation argument of the first goal. Facts are transformed into rules, that call the remaining continuation, denoted as the meta-call Cont.

```
expr(t(T), [t(T)|Xs],Xs, Cont) ←
    Cont.
expr(node(TL,TR), [op|Xs0],Xs, Cont) ←
    expr(TL, Xs0,Xs1, expr(TR, Xs1,Xs, Cont)).
```

2nd step: separation of an output argument. The equation below introduces two new structures expr1/3 and rest/2. These two new function symbols serve as an alternative (and hopefully more efficient) representation for the old function symbol expr/4. By and large, the equation is compiled into the program as follows. In the body of the clauses the equation is used to replace the old function symbol. For every clause containing the expr/4 in the head, we add a new alternative clause. In our example, the program is duplicated.

```
expr(T, Xs0,Xs, Cont) ≒ expr1(T, Xs0, rest(Xs, Cont)).

expr(t(T), [t(T)|Xs], Xs, Cont) ←
    Cont.
expr(node(TL,TR), [op|Xs0],Xs, Cont) ←
    expr1(TL, Xs0, rest(Xs1, expr1(TR, Xs1, rest(Xs, Cont)))).

expr1(t(T), [t(T)|Xs], rest(Xs, Cont)) ←
    Cont.
expr1(node(TL,TR), [op|Xs0], rest(Xs, Cont)) ←
    expr1(TL, Xs0, rest(Xs1, expr1(TR, Xs1, rest(Xs, Cont)))).
```

3rd step: folding of the redundant definition. The definition of expr/4 is expressed with the help of expr1/3. This step serves only to undo the duplication of code. For a practical transformation system it is indeed easier to combine steps 2 and 3 to a single transformation step.

[2]To be more precise, the strategy presented by Proietti and Pettorossi [21] is able to remove the existential variable, but introduces a new, different existential variable.

expr(T, Xs0,Xs, Cont) ←
    expr1(T, Xs0, rest(Xs, Cont)).

expr1(t(T), [t(T)|Xs], rest(Xs, Cont)) ←
    Cont.
expr1(node(TL,TR), [op|Xs0], rest(Xs, Cont)) ←
    expr1(TL, Xs0, rest(Xs1, expr1(TR, Xs1, rest(Xs, Cont)))).

4th step: simplification of the continuation. The structure rest(Xs,Cont) is redundant in the rule. It does not contribute anything to the computation. This can be seen from the equation above: rest(Xs,Cont) will always occur where expr1/3 occurs. It is therefore safe to generalize the second clause in expr1/3. This generalization does not require a global analysis of the program.

expr(T, Xs0,Xs, Cont) ←
    expr1(T, Xs0, rest(Xs, Cont)).

expr1(t(T), [t(T)|Xs], rest(Xs, Cont)) ←
    Cont.
expr1(node(TL,TR), [op|Xs0], **RestXsCont**) ←
    expr1(TL, Xs0, rest(Xs1, expr1(TR, Xs1, **RestXsCont**))).

5th step: definition of a separate predicate to execute the continuation. To make the continuations in expr1/3 more uniform we define a new predicate demo_rest/2 that deals with the continuation rest/2. Again, as step 3, this step serves only to keep the program compact. When transforming several predicates this step helps to keep the program compact. For every predicate a single clause is added for demo_rest/2. Without this step, step 6 would expand every fact of the original program to several rules.

expr(T, Xs0,Xs, Cont) ←
    expr1(T, Xs0, rest(Xs, Cont)).

expr1(t(T), [t(T)|Xs], **RestXsCont**) ←
    demo_rest(RestXsCont, Xs).
expr1(node(TL,TR), [op|Xs0], RestXsCont) ←
    expr1(TL, Xs0, rest(Xs1, expr1(TR, Xs1, RestXsCont))).

demo_rest(rest(Xs, Cont), Xs) ←
    Cont.

6th step: compaction of the continuation. The existential variable Xs1 occurs only in rest/2 and expr1/3. A new structure rest_expr1/3 is introduced to combine both. Xs1 is therefore reduced to a void variable. demo_rest/2 contains new clause in order to read the new representation rest_expr1/3.

rest(Xs1, expr1(TR, Xs1, RestXsCont)) ≒ **rest_expr1(**TR, Xs1, RestXsCont).

expr(T, Xs0,Xs, Cont) ←
    expr1(T, Xs0, rest(Xs, Cont)).

expr1(t(T), [t(T)|Xs], RestXsCont) ←
    demo_rest(RestXsCont, Xs).
expr1(node(TL,TR), [op|Xs0], RestXsCont) ←
    expr1(TL, Xs0, **rest_expr1(TR, _Xs1, RestXsCont)**).

demo_rest(rest(Xs, Cont), Xs) ←
    Cont.
demo_rest(rest_expr1(TR, Xs, RestXsCont), Xs) ←
    expr1(TR, Xs, RestXsCont).

7th step: elimination of the void existential variable. Since Xs1 is now a void variable we reduce it completely with the equation below. Note, that this equation does not preserve equivalence between general terms. It has to be ensured that the equation is applied (as in this example) only to void variables and that the corresponding simplified structure rest_expr2/2 is read only once. In the case of continuations this is trivially true. In a transformation system it is useful to combine steps 6 and 7.

rest_expr1(TR, _Xs1, RestXsCont) $\doteq$ **rest_expr2(**TR, RestXsCont).

```
expr(T, Xs0,Xs, Cont) ←
    expr1(T, Xs0, rest(Xs, Cont)).

expr1(t(T), [t(T)|Xs], RestXsCont) ←
    demo_rest(RestXsCont, Xs).
expr1(node(TL,TR), [op|Xs0], RestXsCont) ←
    expr1(TL, Xs0, rest_expr2(TR, RestXsCont)).

demo_rest(rest(Xs, Cont), Xs) ←
    Cont.
demo_rest(rest_expr2(TR, RestXsCont), Xs) ←
    expr1(TR, Xs, RestXsCont).
```

2.2 Arbitrary grammar rules

The transformation above is easily extended to optimize arbitrary translated DCG-clauses, or as a further generalization EDCGs [22].

- For every binary predicate $p/n+2$ an equation is added to split the rest list from the other arguments (as in step 2): $p(a_1, ...a_n$, Xs0, Xs, Cont) $\doteq$ $p\prime(a_1, ...a_n$, Xs0, rest(Xs, Cont)). It is important to note that all equations must contain *the same* function symbol rest/2.
- The definition of the interface rule as in step 3 can be avoided if all calls to phrase/3 are known statically.
- Simplification as in step 4.
- The definition of demo_rest/2 is the same as above. I.e., there is only a single auxiliary predicate introduced, independent of the number of different predicates.
- For every function symbol $p\prime/n+1$ introduced above, a new equation is added: rest(Xs, $p\prime(a_1, ...a_n$, Xs, XsCont)) $\doteq$ rest_$p\prime(a_1, ...a_n$, XsCont).

To summarize, every grammar rule is translated to a single new clause, a single auxiliary predicate is defined, that serves as the 'meta-call' within the grammar.

2.3 Optimizations performed

The final program in our example above shows most of the optimizations performed by EBC on general grammar rules. All these optimizations can be observed on existing Prolog systems based on the WAM.

1. It contains a smaller continuation. Instead of five memory cells only three are used. I.e., all memory used to represent the difference list is saved.

2. The program analyses the first 2551 phrases 67% faster with BinProlog 1.39 and 30% faster with the compiler SICStus Prolog 2.1. Even though the original program uses SICStus' environment stack and the transformed program uses the heap the transformed version is faster. Both systems were measured on SPARCstation ELC, CPU 33MHz Cypress, 8Mb RAM.

3. The number of argument registers is reduced by one, therefore reducing the size of choice points.

4. All trail checks related to passing the difference list further on are omitted.

5. In the case that occur-checks are desired, all occur-checks related to the difference list in the original program are eliminated up to a single occur-check for every solution of the goal :- expr(T, Xs0, Xs). For the goal ← expr(T, Xs, []) *no* occur-check has to be executed for handling the difference list. Note, that one of the arguments against the occur-check is that it reduces Prolog's efficiency in handling difference lists.

   Joachim Beer's extension to the WAM [3, 4] designed to reduce the number of trails and occur-checks should be seen as a rather complementary approach to occur-check reduction since his machine cannot handle difference lists in the same manner. In particular, in the program quicksort/3 as defined in [4], page 54, we are able to remove all of the 50 necessary occur-checks, while Beer's machine still performs 49.

2.4 Application to DCGs and similar formalisms

The example above presented a typical case occurring in many programs. By using difference lists a state is propagated further. In the case of DCGs the state is the string to be analyzed or generated. With the help of EBC-transformations these states do not require space within the continuation as long as all predicates share the difference list. The following table shows the gain in space consumption of our transformation for generalized DCGs (like EDCGs [22]) that are also capable of including 'external predicates' (e.g. for symbol table lookup) within the grammar. These generalized DCGs posses n implicit states. The gain for 'ordinary' DCGs, where only a single simple state is present, is given with $n = 1$. Our transformation has an additional initialization cost for implementing the entry predicate phrase. However, all continuations within the grammar are either reduced in size or equal (when using external predicates heavily). In particular, in the usual case (case 5), the size of the continuation is reduced considerably.

The additional states in an EDCG can be used for tasks currently implemented with side effects or manual transformations. E.g.: a) error recovery b) line counting c) indentation checks d) safe left recursions. With the help of our transformation, the cost for such additional arguments is very small, comparable to the cost of global variables in procedural languages.

	situation	gain
1 orig. EBC	$\leftarrow$ phrase(g,x_1,...,x_n, y_1,...,y_n). 0 $(2+n)$	$-(2+n)$
2 orig. EBC	$\leftarrow$ phrase(g,x_1,...,x_n,[],...,[]). 0 (2)	-2
3 orig. EBC	$\leftarrow$ ..., phrase(g,x_1,...,x_n, y_1,...,y_n). $(2+n_g+2n)$ $(2+n_g+n)+(2+n)$	-2
4 orig. EBC	$\leftarrow$ ..., phrase(g,x_1,...,x_n,[],...,[]). $(2+n_g+2n)$ $(2+n_g+n)+(2)$	$n-2$
5 orig. EBC	$\ldots \longrightarrow \ldots, g, \ldots$ $(2+n_g+2n)$ $(2+n_g)$	$2n$
6 orig. EBC	$\ldots \longrightarrow \{\ldots\}.$ $(2+2n)$ $(2+n)$	n
7 orig. EBC	$\ldots \longrightarrow \{\ldots\}, g, \ldots$ $(2+n_g+2n)$ $(2+n_g+n)$	n
8 orig. EBC	$\ldots \longrightarrow \ldots, g, \{\ldots\}.$ $(2+n_g+2n)+(2+2n)$ $(2+n_g)+(2+n)+(2+n)$	$2n-2$
9 orig. EBC	$\ldots \longrightarrow \ldots, g, \{\ldots\}, h, \ldots$ $(2+n_g+2n)+(2+n_h+2n)$ $(2+n_g)+(2+n)+(2+n_h+n)$	$2n-2$

3 EBC-transformation

EBC-transformations transform a given binary program into an equivalent one. We do not consider currently non-binary programs. The transformation formalism is divided into three parts: equations providing alternative representations, compilation of these equations into the program, simplification of the compiled programs.

In the sequel we make no distinction between predicate symbols and function symbols. A binary program can be represented by a single binary predicate, encoding all predicate symbols with function symbols. We use the word continuation for the head and the (single) goal of a binary clause. If a continuation is a function symbol it contains (beside others) a single argument that is used to hold further subcontinuations. Usually this argument is the last one. We denote by T_{old}, the set of terms constructed from the function symbols F_{old} that are present in the original program and some variables. $T_{new} \supset T_{old}$ denotes the corresponding set of the new program. θ and σ are substitutions.

3.1 Conservative extension

We define a *conservative extension to syntactic unification* as a set E of equations of the form '$x \doteq y$' with $x, y \in T_{new}$ such that for all terms s and $t \in T_{old}$:

$$\exists\theta.s\theta = t\theta \text{ iff } \exists\rho.s\rho =_E t\rho$$ [3]

[3] $=_E$ means equality modulo the equations E.

As long as the equations in E contain only terms in T_{old} such an extension is considered trivial. No alternative representations could be constructed.

Our notion of conservative extension can be seen as a restriction to the notion of consistency in rewrite systems and algebraic specifications [10]. A set of equations E is consistent if for all *ground* terms the condition above holds. I.e., consistency is a necessary but not sufficient condition to qualify E as a conservative extension. The following example gives an equation that qualifies as consistent but not as a conservative extension.

$$F_{old} = \{f, g, c\}, \quad F_{new} = \{h\}, \quad E = \{f(X) \doteq g(h(X))\}$$

The equation E is consistent with respect to F_{old}. On the other hand, E is not a conservative extension since there is no substitution θ for $f(c) = g(X)\theta$ but there is a $\rho = \{Y \mapsto h(c)\}$ with $f(c)\rho =_E g(Y)\rho$.

3.2 Compilation of equations

We are interested in implementing the equational unification above with the help of syntactic unification. Since we want to keep the overhead for compiling equations as low as possible we only compile equations over terms that are never variables at runtime. To avoid any dataflow analysis we restrict ourselves to the continuation generated by the transformation from Prolog to binary Prolog. The compilation is divided into the compilation of goals and heads.

Compilation of goals. Continuations in the goals are never read but are simply written. We are therefore free to replace any subcontinuation that matches a given equation by the other side of the equation. Let r be a subcontinuation, and $s \doteq t$ an equation. The subcontinuation r can be replaced by $t\theta$ if $r = s\theta$. Remark that we are allowed to use the equations in any direction desired.

As a special case, equations $s \doteq t$ with $\mathrm{VAR}(s) \subset \mathrm{VAR}(t)$ are allowed to eliminate void variables. In this case, all void variables in the equation must match void variables in the subcontinuation.

Compilation of heads. The head of a clause reads and unifies continuations. It must therefore be able to deal with all alternative representations of a term. For every clause C we create for every rewriting yielding a different head a new clause C_i. All resulting clauses C_i must not be unifyable with one another. Usually, only a single transformation step is required for every combination of a clause and an equation. The rewriting of clauses must not necessarily terminate. This is in particular the case when recursive equations are used. If a clause can be rewritten for the second time the term to be rewritten is separated into a new auxiliary predicate. In this manner, simple infinite rewritings can be compiled into recursive predicates.

3.3 Simplification of clauses

In most of the interesting applications of EBC investigated so far a simplification step is required after the compilation of equations. In this step redundancies in the program are removed that have been made explicit by the introduced equations. The conditions for simplification depend only on the equations E

compiled in the previous step and the clause to be simplified. No global analysis is required to validate the simplification step.

The original clause $C_o = H_o \leftarrow B_o$ was compiled into $C = H \leftarrow B$, besides possibly other clauses. Simplification is carried out by generalizing a binary clause $C = H \leftarrow B$ to a clause $C_g = H_g \leftarrow B_g$ under the following conditions:

1. $C = C_g\theta$, and $\mathrm{dom}(\theta) \subseteq \mathrm{VAR}(B_g)$. I.e., only those generalizations of the head are allowed, that are covered by generalizations in the body.

2. For every clause $H \leftarrow B$ and its generalization(s) $H_g \leftarrow B_g$ the following condition must hold for all $i, j \geq 1$:

   $\mathrm{Old}(P_i(H) \leftarrow P_j(G)) = \mathrm{Old}(P_i(H_g) \leftarrow P_j(G_g))$

   $\mathrm{Old}(H \leftarrow B)$ is the set of rules $H_o \leftarrow G_o$ constructed with T_{old} that are unifiable with $H \leftarrow B$.

   $P_i(L)$ is a projection of the continuation L that substitutes the i-th subcontinuation of L by a new free variable $Cont_i$. $P_1(L) = Cont_1$, i.e., the whole term is substituted. $P_2(L)$ substitutes the first subcontinuation by $Cont_2$ etc.

   This condition ensures that during execution the bindings at the outer continuations will be identical to the original program. This means that built-in predicates, read and write statements, cuts etc. may be used in the programs to be transformed at any place[4].

Examples. Given the equation p(E, Cont) $\doteqdot$ q(E, r(E, Cont)), the clause p(E, Cont) $\leftarrow$ q(E, r(E, Cont)) can be generalized to p(E, Cont) $\leftarrow$ q(E, r(_E, Cont)). However, it is not allowed to generalize to p(E, Cont) $\leftarrow$ q(_E, r(E, Cont)).

Note, that simplification is driven heavily by the redundancy exposed in the equations E:

```
p(X,X, Cont) ←
    q(X,X, Cont).

p(A,B, Cont) ≒ p1(r(A,B, Cont)).
q(A,B, Cont) ≒ q1(r(A,B, Cont)).

p1(r(A,A, Cont)) ←
    q1(r(A,A, Cont)).
```

The following generalization is invalid, since Old(p1(ABCont) $\leftarrow$ p1(ABCont)) contains the clause p(X,Y, Cont) $\leftarrow$ q(X,Y, Cont) while Old(p1(r(A,A, Cont)) $\leftarrow$ q1(r(A,A, Cont)) does not contain this clause.

```
p1(ABCont) ← % violates Cond. 2
    q1(ABCont).
```

However, the same generalization is valid, under a different set of equations:

```
p(A,A, Cont) ≒ p2(r(A,A, Cont)).
q(A,A, Cont) ≒ q2(r(A,A, Cont)).

p2(r(A,A, Cont)) ←
    q2(r(A,A, Cont)).

p2(ABCont) ← % valid
    q2(ABCont).
```

[4]Cuts are represented in binary Prolog with an auxiliary variable used for labeling.

4 Transformation of context arguments

The application of the generalization rules is demonstrated by the following program that uses the built-in predicate var/1 that is sensible to bindings. While the goal var/1 is part of the body, we will still write it as a separate goal to ease reading.

```
equalnodes(nil, _El).
equalnodes(node(El,L,R), El) ←
    var(El),
    equalnodes(L, El),
    equalnodes(R, El).

equalnodes(nil, _El, Cont) ←
    Cont.
equalnodes(node(El,L,R), El, Cont) ←
    var(El),
    equalnodes(L, El, equalnodes(R, El, Cont)).
```

The continuation contains the redundant variable El. We make this redundancy more explicit by duplicating the occurrences of El.

```
equalnodes(T, El, Cont) ≒ equalnodes1(T, El, rest(El, Cont)).

equalnodes(T, El, Cont) ←
    equalnodes1(T, El, rest(El, Cont)).

equalnodes1(nil, El, rest(El, Cont)) ←
    Cont.
equalnodes1(node(El,L,R), El, rest(El, Cont)) ←
    var(El),
    equalnodes1(L, El, rest(El, equalnodes1(R, El, rest(El, Cont)) )).
```

In the following step several variables are renamed.

```
equalnodes(T, El, Cont) ←
    equalnodes1(T, El, rest(_El, Cont)).

equalnodes1(nil, El, rest(El, Cont)) ←
    Cont.
equalnodes1(node(El,L,R), El, rest(El2, Cont)) ←
    var(El),
    equalnodes1(L, El, rest(El1, equalnodes1(R, El1, rest( El2, Cont)) )).
```

Note that the e.g.,following generalizations are invalid:

```
% wrong example
equalnodes(T, El, Cont) ←
    equalnodes1(T, _El, rest(El, Cont)). % violates Cond. 2

equalnodes1(nil, _El1, rest(_El2, Cont)) ← % violates Cond. 1
    Cont.
equalnodes1(node(El,L,R), El2, rest(El1, Cont)) ←
    var(El),
    equalnodes1(L, El2, rest(El1, equalnodes1(R, El1, rest(El, Cont)) )).
% El2 and El violate Cond. 4
```

The redundant subcontinuation is removed in the clause equalnodes1/3.

```
equalnodes(T, El, Cont) ←
    equalnodes1(T, El, rest(_El, Cont)).

equalnodes1(nil, El, rest(El, Cont)) ←
    Cont.
equalnodes1(node(El,L,R), El, ElCont) ←
    var(El),
    equalnodes1(L, El, rest(El1, equalnodes1(R, El1, ElCont) )).
```

The subsequent steps are similar to our first example expr/4. We omit the intermediary steps, only showing the result and the equations used.

```
rest(El, equalnodes(R, El, ElCont)) ≒ rest_equalnodes(R, ElCont).
rest(_El, Cont) ≒ rest1(Cont).

equalnodes(T, El, Cont) ←
    equalnodes1(T, El, rest1(Cont)).

equalnodes1(nil, El, ElCont) ←
    restel(ElCont, El).
equalnodes1(node(El,L,R), El, ElCont) ←
    var(El),
    equalnodes1(L, El, rest_equalnodes1(R, ElCont)).

restel(rest1(Cont),_El) ←
    Cont.
restel(rest_equalnodes(R, ElCont),El) ←
    equalnodes(R, El, ElCont).
```

The final program does no more contain the variable El in its continuation. Furthermore, El resides always in the second argument. Executed on the WAM, the variable El can be considered as being allocated 'globally', (i.e.: interprocedurally) in the second argument register.

5 Strategies and schemes of equations

For a transformation system to be of practical use, strategies avoiding the very large search space have to be developed. There are several sources that contribute to the search space of EBC-transformations: a) the choice of an appropriate scheme of equations b) the effective instantiation of the scheme c) the choice of simplifications.

The development of schemes of conservative equations seems to be an inherent manual operation. The methods developed within the context of rewrite systems and algebraic specification might be adaptable to automate this process. However, a scheme of equations often determines a particular optimization. E.g. optimization of difference lists, context arguments. It seems that a practical transformation system might only choose from given schemes of equations; similar to a compiler for an imperative programming language that comprises several optimization passes like common subexpression elimination, global or even interprocedural register allocation etc. Below, we present the equational schemes and their use in optimization.

The effective instantiation of an equation scheme is often 'driven' by the subsequent simplification. Since the conditions for simplification are closely related to the equations used, we can on the other hand take the simplifications as a condition for the application of a certain equation. For example, in predicate expr/4 the equation expr(T, Xs0,Xs, Cont) ≒ **expr1(**T, Xs0, **rest(**Xs, Cont)) was the only choice possible to allow a subsequent simplification of the continuation rest(Xs, Cont). If we would have tried expr(T, Xs0,Xs, Cont) ≒ **expr2(**T, Xs, **rest(**Xs0,Cont)). instead, no simplification of rest(Xs, Cont) would have been possible. The search for an appropriate equation can therefore be pruned by the subsequent simplification. It seems therefore appropriate to associate to every equation specialized simplification operations thus avoiding the search space within simplification.

For the sake of simplicity, we present concrete equations for the encountered schemes, the arguments of function symbols might be appropriately increased.

Output argument splitting. An argument Out is only passed further from the head to the last goal in a clause. Typically, the second argument of a difference list or an accumulator is an output argument.

old1(Args,Out,Cont) $\doteqdot$ new1(Args,new(Out, Cont))
old2(Args,Out,Cont) $\doteqdot$ new2(Args,new(Out, Cont))
...

Simplification: removal of new(B, Cont)

Forced output argument splitting. A context argument Ctx is passed around, by duplicating its occurrences it can be treated as an output argument above.

old1(Args, Ctx, Cont) $\doteqdot$ new1(Args, Ctx, new(Ctx, Cont))
old2(Args, Ctx, Cont) $\doteqdot$ new2(Args, Ctx, new(Ctx, Cont))
...

Simplification: renaming of duplicated variables, removal of new(Ctx, Cont).

Compacting continuations. Two continuations that share some arguments, or variables are folded into a new one. The new structure contains the union Args12 of both arguments.

old1(Args1, old2(Args2, Cont)) $\doteqdot$ new(Args12, Cont).

No simplification required. However, most occurrences of old1/2 should have been replaced. Note that Args1 and Args2 may contain function symbols oldf/1 that are not continuations themselves. e.g.:

old1(A, old2(oldf(A), Cont)) $\doteqdot$ new(A, Cont).

Recursive context introduction. Used to split recursive programs into several iterations.

old1(A, old2(B, Cont)) $\doteqdot$ old1(A, new(A, old2(B, Cont))).

Simplification: renaming of duplicated variables.

Merging/splitting user continuations. In a program a recursive data structure is used in a 'continuation like' manner. By the following equations, such data structures can be merged with the system continuation. No analysis is required to ensure that a user data structure is 'continuation like'. For, if it is not, not all occurrences can be merged with the system continuation. Typical examples for this scheme merge a list with the continuation; or split recursive parts of a continuation in order to implement them with a counter. See [20] for an exemplary use.

old1(oldc,old2(Cont)) $\doteqdot$ newc(Cont).
old1(oldf(F), Cont) $\doteqdot$ newf(old1(F,Cont)).

Condition of application: all occurrences of oldc/0 and oldf/1 should have been replaced.

6 Related work

Source-to-source transformations. Sato and Tamaki's CPS-conversion [23] separates input and output arguments, yielding binary programs. However, their method has to rely on a previous analysis. Otherwise, they do not preserve finite failure and infinite loops. The derived program perm/2 in [23] loops for ← perm(_,const) while the original fails. Further, as observed by Tarau [25], clause indexing is not preserved.

Particular strategies within fold/unfold have been investigated. Proietti and Pettorossi present a fold/unfold-strategy [21] to remove existential variables. Such variables cannot be removed by EBC and vice versa. A fold/unfold-strategy to remove unnecessary structures is presented by Gallagher and Bruynooghe [9]. Demoen [7] considers transformations for binary programs that are explicable within the framework of fold/unfold-transformations. I.e., folding of goals in order to 'build up continuations incrementally' (Sect. 1), partial evaluation (Sect. 2.1); void variable elimination in the body (Sect. 2.3), register move optimization (Sect. 2.5).

Low-level optimizations. Beer [4] optimizes uninitialized variables. Meyer uses destructive assignments in environments [17]. Compile time garbage collection [14, 18, 11] might yield similar results for difference lists. However, no published results on programs like DCGs are known to the author.

Other formalisms. The compilation of unification in EBC-transformations is related to narrowing ([2] 6.1.3). Grammars in λ-Prolog [15] do not need any auxiliary variables. Thus λ-Prolog is an interesting formalism for further transformations. CPS-transformations were originally investigated in functional languages [26]. We have so far not found transformations corresponding to EBC in functional languages. The reason for this seems to be the absence of the logical variable. The close relation of Attribute Grammars and logic programs [8] suggests that optimizations for AGs [13, 12] can be applied to Prolog; as investigated by Bouquard [5] comparing WAM and FNC-2.

7 Further work

The further development of strategies seems to be most promising since EBC-transformations are able to obtain programs that are very close to their counterparts in procedural programming languages. Although our transformation rules are equivalence preserving they are not completely invertible. In particular, the simplification of continuations cannot be inverted. This means, that for certain transformations, we have to 'invent' the desired program and then derive from the invented program the original program. Further research is required to improve the transformation rules.

The precise relation between fold/unfold for logic programs and EBC is not yet clear to the author. It seems, however, that fold/unfold does not cover EBC-transformations. For example, the existential variable in expr/4 has resisted the author's attempts for removal within fold/unfold.

Our transformation circumvents the environment stack of the classic WAM. However, WAM compilation can be easily adopted compiling the continuations back into stack frames. Registers would be valid beyond the 'proceed barrier'.

Acknowledgements. I thank Gernot Salzer for many comments on EBC-transformations. The three anonymous referees provided many appreciated comments.

References

[1] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

[2] F. Baader and J. Siekmann. Unification theory. In D. Gabbay, C. Hogger, and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*. Oxford University Press, Oxford, UK, 1993. To appear.

[3] J. Beer. The occur-check problem revisited. *The Journal of Logic Programming*, 5(3):243–262, September 1988.

[4] J. Beer. *Concepts, Design, and Performance Analysis of a Parallel Prolog Machine*, volume 404 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[5] J.-L. Bouquard. *Etude des rapports entre Grammaire Attribuées et Programmation Logique: Application au test d'occurrence et à l'analyse statique*. PhD thesis, Université d'Orleans, 1992.

[6] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.

[7] B. Demoen. On the transformation of a Prolog program to a more efficient binary program. Technical Report 130, K.U.Leuven Department of Computer Science, revised version LOPSTR92, 1992.

[8] P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.

[9] J. Gallager and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Metaprogramming in Logic*, pages 229–244. K.U. Leuven, Department of Computer Science, Apr. 1990.

[10] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[11] G. Hamilton and S. Jones. Compile-time garbage collection by necessity analysis. In S. P. Jones, G. Hutton, and C. Holst, editors, *Functional Programming, Glasgow 1990*, pages 66–70. Springer-Verlag, London, 1991.

[12] C. Julié and D. Parigot. Space optimization in the FNC-2 attribute grammar system. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications (WAGA)*, volume 461 of *Lecture Notes in Computer Science*, pages 29–45. Springer-Verlag, Sept. 1990.

[13] U. Kastens and M. Schmidt. Lifetime analysis for procedure parameters. In B. Robinet and R. Wilhelm, editors, *Proceedings of the 1st European Symposium on Programming (ESOP '86)*, volume 213 of *Lecture Notes in Computer Science*, pages 53–69. Springer-Verlag, Mar. 1986.

[14] F. Kluźniak. Compile time garbage collection for Ground Prolog. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1490–1505, Seatle, 1988. ALP, IEEE, The MIT Press.

[15] S. Le Huitouze, P. Louvet, and O. Ridoux. Logic grammars and λ-Prolog. In D. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 64–79, Budapest, Hungary, 1993. The MIT Press.

[16] J. Lloyd and J. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[17] M. Meier. Recursion vs. iteration in Prolog. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 157–169, Paris, France, 1991. The MIT Press.

[18] A. Mulkers, W. Winsborough, and M. Bruynooghe. Analysis of shared data structures for compile-time garbage. In D. H. D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, 1990. The MIT Press.

[19] U. Neumerkel. Specialization of Prolog programs with partially static goals and binarization, Dissertation. Bericht TR 1851-1992-12, Institut für Computersprachen, Technische Universität Wien, 1992.

[20] U. Neumerkel. Une transformation de programme basée sur la notion d'équations entre termes. In *Secondes journées francophones sur la programmation en logique (JFPL'93)*, Nîmes-Avingnon, France, 1993.

[21] M. Proietti and A. Pettorossi. Unfolding-definition-folding in this order, for avoiding unnecessary variables in logic programs. In J. Małuszyński and M. Wirsing, editors, *Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 347–358, Passau, Germany, Aug. 1991. Springer-Verlag.

[22] P. V. Roy. A useful extension to Prolog's Definite Clause Grammar notation. *SIGPLAN notices*, 24(11):132–134, Nov. 1989.

[23] T. Sato and H. Tamaki. Existential continuation. *New Generation Computing*, 6(4):421–438, 1989.

[24] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Second International Logic Programming Conference*, pages 127–138, Uppsala, 1984.

[25] P. Tarau and M. Boyer. Elementary logic programs. In P. Deransart and J. Małuszyński, editors, *Programming Languages Implementation and Logic Programming*, volume 456 of *Lecture Notes in Computer Science*, pages 159–173, Linköping, Sweden, Aug. 1990. Springer-Verlag.

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