Continuation Prolog: A new intermediary language for WAM and BinWAM code generation

Ulrich Neumerkel
Technische Universität Wien, Institut für Computersprachen
A-1040 Wien, Austria ulrich@mips.complang.tuwien.ac.at

Abstract

We present a new intermediary language suitable for program transformations that fills the gap between Prolog source level and lower representations like binary Prolog or WAM-code. With the help of Continuation Prolog transformations on the level of continuations can be expressed that are unavailable in the usual settings of WAM-code generation.

1 Introduction

Current implementations of Prolog still lack many optimizations even for applications like syntax analysis. The following grammar rule illustrates the problem.

expr $\longrightarrow$ [op], expr, expr.

In a procedural language such a rule can be implemented with recursive descent. The input string is a global variable which is updated destructively.

$$expr(\ [op|\overbrace{Xs0}],\overbrace{Xs}) \leftarrow \underbrace{expr(\overbrace{Xs0},\overbrace{Xs1}),expr(\overbrace{Xs1},Xs)}_{Xs}).$$

In Prolog however the variable representing the input string is simulated with several different logic variables since logic variables can only represent a single state. These overheads can be observed in all Prolog machines: the variables are allocated in memory; memory must be initialized, the binding of theses variables require superfluous trail checks. Eventually, the different states of the input string are written into memory and read back. This overhead is necessary even if the input string is simply passed around. The procedural counterpart requires for the comparable task a single memory location to represent the variable. Overheads for maintaining the input string only occur if the input string is read. Evidently the imperative implementation is more efficient than a Prolog implementation. For this reason language extensions have been proposed for Prolog to allow destructive updates. With such extensions, however, desirable properties of Prolog like referential transparency are affected.

In this article we present an intermediary language called Continuation Prolog and its transformation system — EBC-transformations. With them we are able to remove all auxiliary variables necessary to represent list differences and similar constructs. The resulting programs can transformed either into binary Prolog (and eventually into BinWAM code) or into WAM-code. Earlier formulations of EBC-transformations were only able to generate optimized binary Prolog [6].

Our transformation overcomes some of the well known pitfalls of the WAM. In particular the lifetime of registers is significantly extended. In the classical WAM setting no argument register remains valid over the proceed instruction. While this problem is well known in the literature, there are only few approaches that address this problem. Two alternate abstract machines have been developed to circumvent the "WAM argument register bottleneck": the VAM [3] and the NTOAM [14] but both introduce also new problems absent in the WAM.

Overview. In Sect. 2 we discuss various implementation strategies and identify in Sect. 3 the rôle of Continuation Prolog. Sect. 4 presents the transformation system. An example demonstrating the benefits of our transformation is given in Sect. 5. WAM code generation for continuation Prolog is discussed in 6.

2 Implementation strategies

Prolog ⇒ WAM (⇒ Machine code). The most widely used approach translates Prolog into the intermediary code of an abstract machine. A further translation step compiles the intermediary code into machine code. Often the WAM-instruction set has been extended and refined considerably. In particular systems for native code generation prefer a more refined level (BAM [8] and PARMA [12]).

Prolog ⇒ Binary Prolog ⇒ reduced WAM. Prolog is first translated into a subset called binary Prolog. Only this representation is used for further translation. Since binary Prolog encodes AND-control with the help of terms, the instruction set required is reduced. All instructions with environment operands can be eliminated. On the other hand, heap consumption increases. The first Prolog implementation based on this approach (BinProlog) has shown that it is a viable alternative to the well established paths of WAM-implementations. BinProlog is comparable in efficiency and significantly smaller than comparable Prolog implementations. The core of BinProlog uses 43 WAM-instructions which are divided into 23 elementary and 20 folded instructions. In contrast SICStus-Prolog, a system based on traditional WAM-technology, uses 500 (folded) instructions for comparable efficiency and functionality. (A comparison of the total size of both systems cannot be made because the two systems provide different functionality.)

Binary Prolog not only simplifies an implementation but allows source-to-source transformations that are not possible on the original source code. In [4] EBC-transformations were presented that allow many optimizations on the source level of binary Prolog like interprocedural register allocation and removal of all auxiliary variables used to represent list differences as they occur in translated DCG-rules.

Our approach of using binary Prolog as an intermediary language for EBC-transformations has however several practical drawbacks. First, when using binary Prolog for source-to-source optimizations several complications arise: While —in principle– all predicate arguments are treated in the same manner, a distinction between the continuation argument and other arguments is necessary. Handling this separate argument complicates the implementation of transformation rules considerably. Second, binary Prolog cannot be used with ease as an intermediary language for WAM-generating systems. In binary Prolog all environments are already encoded as terms. But in a WAM-based system environments should be implemented differently using the WAM's environment stack.

To overcome these drawbacks a new formalism was developed — Continuation Prolog. EBC-transformations are now defined on the level of Continuation Prolog. The resulting improved programs can then be translated either into Binary Prolog or directly into full WAM code.

3 Continuation Prolog

Since continuations are not treated in the same manner as ordinary arguments during program transformation our new formalism avoids an explicit representation of continuations. Rules in Continuation Prolog are of the form (Head ⇜ Body), where both Head and Body are sequences (lists). The translation of a Prolog program to Continuation Prolog is straight forward. The head of a rule is mapped onto a list with a single element. The rule body maps onto a list of goals. Only cuts must be treated differently. They are implemented, similar to binary Prolog with the help of a labeled cut. Facts are treated like rules with an empty body.

```
                                [ expr([z|Xs],Xs) ] ⇜
expr([z|Xs],Xs).                    [].
                                [ expr([op|Xs0],Xs) ] ⇜
expr([op|Xs0],Xs) ←                 [ expr(Xs0,Xs1),
   expr(Xs0,Xs1),                     expr(Xs1,Xs) ].
   expr(Xs1,Xs).
```

The meaning of a program in Continuation Prolog is defined by the meta interpreter contint/1. The predicate contint/1 is a generalization of an ordinary Prolog meta interpreter. For programs that have been directly translated from regular Prolog, the meta interpreter is equivalent to a meta interpreter for regular Prolog. In this case the rule heads will always contain a list with a single element and instead of ruledif f/2 the specialized predicate ruledif f_spez/2 can be

used.

```
contint(C) ←
   contint([C],[]).

contint(Cs,Cs).
contint([C|Cs0],Cs) ←
   rulediff([C|Cs0], Cs1),
   contint(Cs1,Cs).
```

```
rulediff(Cs0, Cs) ←
   ( Hs ↞ Gs ),
   append(Hs, Cs1, Cs0),
   append(Gs, Cs1, Cs).
rulediff_spez([H|Cs1], Cs) ← % regular meta interpreter
   ( [H] ↞ Gs ),
   append(Gs, Cs1, Cs).
```

The only difference between Continuation Prolog and regular Prolog is that a rule head can contain more than one element. An inference with rulediff/2 is thus able to read several elements (with the goal append(Hs, Cs1, Cs0)). This extension allows transformations that cannot be expressed with traditional transformation systems which operate on the level of goals (fold-/unfold according to Sato and Tamaki [10]).

The relation between Continuation Prolog and Binary Prolog is straight forward. Each element C of a continuation Cs in Continuation Prolog corresponds to a term BinCont in Binary Prolog. The mapping via contclause_to_binaryclause/2 sheds some light on the difficulties when using Binary Prolog for program transformations since continuations must be manipulated via =../2.

```
contlit_to_binlitsdiff(C, BinCont0,BinCont) ←
   C =.. [F|Args],
   append(Args,[BinCont],ArgsBinCont),
   BinCont0 =.. [F|ArgsBinCont].

contlits_to_binlitsdiff([], BinCont,BinCont).
contlits_to_binlitsdiff([C|Cs], BinCont0,BinCont) ←
   contlit_to_binlitsdiff(C, BinCont0,BinCont1),
   contlits_to_binlitsdiff(Cs, BinCont1,BinCont).

contclause_to_binaryclause((Hs ↞ Gs), (BH ← BG)) ←
   contlits_to_binlitsdiff(Hs, BH, Cont),
   contlits_to_binlitsdiff(Gs, BG, Cont).
```

A subset of Continuation Prolog can be translated directly into WAM-code. In this case the environment stack is used to represent continuations. Heap consumption is now similar to a traditional WAM. In a later section we will identify a subset of Continuation Prolog that can be translated to WAM-code with ease.

4 The transformation system

EBC- (*equality based continuation*) transformations transform a program in Continuation Prolog into an operationally equivalent one. Also infinite derivations and errors are preserved. The transformation formalism is divided into

three parts: equations providing alternative representations for continuations, compilation of these equations into the program, simplification of the compiled programs.

A continuation is a list of continuation elements. A subcontinuation s is a sublist of continuation c where $c = r{\cdot}s{\cdot}t$. The symbol $\cdot$ denotes the concatenation of two continuations. A rule in Continuation Prolog consists of a continuation in the head and a continuation in the body.

4.1 Equations of continuations

The transformation system uses equations to introduce new alternate (and hopefully more efficient) representations for continuations. Equations over continuations are of the form $s \doteq t$. With this equation every continuation c where $c = u \cdot s\theta \cdot v$ is equivalent to d with $d = u \cdot t\theta \cdot v$. For example, the equation

[expr(Xs0,Xs)] =.= [**expr1**(Xs0), **rest**(Xs)].

states that the new functors expr1/1 and rest/1 may serve as substitutes for expr/2.

A set E of equations is called a **conservative extension** if for all continuations s, t, that can be constructed from a given program the equations do not affect unification.

$$\exists\theta.s\theta = t\theta \;\text{ iff }\; \exists\rho.s\rho =_E t\rho$$

Whether a given set of equations is a conservative extension or not is undecidable in general. We are using only a few schemes of equations. The following schemes form conservative extensions:
[old] =.= [new_1, ..., new_n] [old_1, ..., old_n] =.= [new] [old_1, old_2] =.= [old_1, new, old_2]

4.2 Compilation of equations

Equations are implemented with the help of syntactic unification. They are compiled into the program prior to execution. 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. With the continuation equation $u \doteq v$ the body $c_0 = r \cdot s \cdot t$ with $s = u\theta$ is translated into $c_1 = r \cdot v\theta \cdot t$ Remark that we are allowed to use the equations in any direction desired.

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 unifiable with one another. Usually, only a single transformation step is required for every combination of a clause and an equation.

For the existing rule $r \cdot s \twoheadleftarrow t$ the continuation equation $u \cdot v \doteq w$ adds the following rule:

1. The rule $r \cdot s \twoheadleftarrow t$ is rewritten with equation $u \cdot v \doteq w$ if $\exists\theta.s\theta = u\theta$ into the new rule $(r \cdot w \twoheadleftarrow t \cdot v)\theta$

2. The rule $r \cdot s \cdot t \twoheadleftarrow u$ is rewritten with equation $v \doteq w$ if $\exists\theta.s\theta = v\theta$ into the new rule $(r \cdot w \cdot t \twoheadleftarrow u)\theta$.

4.3 Simplification of clauses

In all 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 $C_o = H_o \twoheadleftarrow B_o$ was translated into $C = H \twoheadleftarrow B$. The clause $C = H \twoheadleftarrow B$ can be simplified further on to $C_g = H_g \twoheadleftarrow B_g$ as follows:

Continuation shortening: $H = H_g \cdot X, \quad B = B_g \cdot X$

The common suffix X in head and body is removed. The variable in X must not occur in H_g and B_g.

Generalization of arguments: $C = C_g\theta$ with $\mathrm{dom}(\theta) \subseteq \mathrm{VAR}(B_g)$

i.e., Only those generalizations are permitted that are covered by the body of a clause.

To ensure that a simplification is valid the following condition must hold:

For each clause $H \twoheadleftarrow B$ and its simplification $H_g \twoheadleftarrow B_g$, for all $i, j \geq 1$:

$$\mathrm{Old}(P_i(H) \twoheadleftarrow P_j(B)) = \mathrm{Old}(P_i(H_g) \twoheadleftarrow P_j(B_g))$$

$\mathrm{Old}(H \twoheadleftarrow B)$ is the set of rules $H_o \twoheadleftarrow B_o$ that are unifiable with $H \twoheadleftarrow B$.

P is a projection. $P_i(\mathrm{Cont})$ shortens the continuation starting from the i-th element.

Example: $P_1([k1, k2]) = []$, $P_2([k1, k2]) = [k1]$ 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.

5 Example of an EBC-transformation

Predicate expr/2 describes a list difference of simple prefix expressions. This predicate is the minimal nontrivial example where an existential variable occurs (Xs1) which cannot be removed by fold-/unfold transformations. The fold-/unfold strategy presented by Proietti and Pettorossi [7] is able to remove the existential variable but introduces a new different existential variable.

```
[ expr([z|Xs],Xs) ] ⇜
    [ ].
[ expr([op|Xs0],Xs) ] ⇜
    [ expr(Xs0,Xs1),
      expr(Xs1,Xs) ].
```

Separation of an output argument. The equation below introduces two new structures expr1/1 and rest/1. These two new function symbols serve as an alternative (and hopefully more efficient) representation for the old function symbol expr/2.

```
[ expr(Xs0,Xs) ] =.= [ expr1(Xs0), rest(Xs) ].
```

```
[ expr( Xs0,Xs) ] ⇜
    [ expr1( Xs0),
      rest(Xs) ].
```

```
[ expr1([z|Xs]),
  rest(Xs) ] ⇜
    [ ].
[ expr1([op|Xs0]),
  rest(Xs) ] ⇜
    [ expr1(Xs0),
      rest(Xs1),
      expr1(Xs1),
      rest(Xs) ].
```

Simplification of the continuation. The structure rest(Xs) is redundant in the rule. It does not contribute anything to the computation. This can be seen from the equation above: rest(Xs) will always occur where expr1/1 occurs. It is therefore safe to generalize the second clause in expr1/1. This generalization does not require a global analysis of the program.

```
[ expr1([z|Xs]),
  rest(Xs) ] ⇜
    [ ].
[ expr1([op|Xs0]) ] ⇜
    [ expr1(Xs0),
      rest(Xs1),
      expr1(Xs1) ].
```

Definition of an auxiliary predicate. To keep the program compact, all occurrences of rest/1 in the head are folded into the auxiliary predicate demo/1.

```
[ expr1([z|Xs]) ] ↞          [ demo(Xs),
    [ demo(Xs) ].              rest(Xs) ] ↞
[ expr1([op|Xs0]) ] ↞            [].
    [ expr1(Xs0),
      rest(Xs1),
      expr1(Xs1) ].
```

Compaction of the continuation. The last occurrence of rest/1 in expr1/1 is removed by combining rest/1 and expr1/1. The continuation rest/1 has now an alternate representation: rest_expr1/0. All clauses reading rest/1 are duplicated to read the new representation. In this example demo/1 gets a new clause.

[rest(Xs1), expr1(Xs1)] =.= [**rest_expr1**].

```
[ expr(Xs0,Xs) ] ↞
    [ expr1(Xs0),
      rest(Xs) ].
                             [ demo(Xs),
[ expr1([z|Xs]) ] ↞          rest(Xs) ] ↞
    [ demo(Xs) ].              [].
[ expr1([op|Xs0]) ] ↞        [ demo(Xs),
    [ expr1(Xs0),            rest_expr1 ] ↞
      rest_expr1 ].            [ expr1(Xs) ].
```

The resulting program in binary Prolog has now a smaller continuation than the original predicate and is executed 70% faster on BinProlog. Slightly smaller relative speedups can be obtained when generating WAM-code.

```
expr(Xs0,Xs, Cont) ←
   expr1(Xs0, rest(Xs, Cont)).      demo(rest(Xs, Cont), Xs) ←
                                       Cont.
expr1([z|Xs], Cont) ←               demo(rest_expr1(Cont), Xs) ←
      demo(Cont, Xs).                  expr1( Xs, Cont).
expr1([op|Xs0], Cont) ←
      expr1(Xs0, rest_expr1(Cont)).
```

6 WAM-code generation

To exploit the full WAM instruction set we have to reduce Continuation Prolog to a subset that can be easily translated onto a stack based architecture. The following restrictions allow for a very simple translation scheme.

A WAM-ifiable subset of Continuation Prolog. The WAM-ifiable subset must only contain predicates of the following form:

1. User predicates: They contain heads of a single element. Calls to auxiliary predicates must occur only in clauses with a single goal.

2. Auxiliary predicates: All clauses contain heads of two elements and a body with at most one element. The second element of a head must occur only in a single clause.

All functors of continuation elements must either be predicate functors (occuring as the first element of the head) or auxiliary functors (occuring as the second element in an auxiliary predicate).

In this case the translation is very simple:

- Predicate functors in the body correspond to regular goals.
- Auxiliary functors are translated according to their clause in the auxiliary predicate.
- Calls to auxiliary predicates are mapped onto the "proceed"-instruction.

```
% Prolog code      % Original WAM      % Optimized WAM          % EBC-CP
 — .                — .               [ ifshallow,neck(2),     [ expr(Xs0,Xs) ] ↞
                                        else,endif,
                                        allocate,
                                        get_y_variable(0,1),
                                        init([]),
                                        call(expr1/1,1),          [  expr1(Xs0),
                                        get_y_value(0,0),            rest(Xs) ].
                                        deallocate,
                                        execute(true/0)].
```

```
% Prolog code      % Original WAM            % Optimized WAM      % EBC-CP
expr( [            [ get_list_x0,           [ get_list_x0,       [ expr1([
    z                unify_constant(z),       unify_constant(z),      z
                     unify_x_local_value(1),  neck(1),
    |Xs],            neck(2),                 unify_x_variable(0),    |Xs]) ] ↞
    Xs).             proceed ].               proceed ].              [ demo(Xs) ].
```

```
% Prolog code   % Original WAM          % Optimized WAM      % EBC-CP
expr( [         [ get_list_x0,         [ get_list_x0,       [ expr1([
    op            unify_constant(op),    unify_constant(op),     op
                  neck(2),               neck(1),
                  allocate,              allocate,              |Xs0]) ] ↞
                  get_y_variable(1,1),   unify_x_variable(0),
    |Xs0],Xs) ←   unify_x_variable(0),
                  put_y_variable(0,1),
                  init([]),              init([]),
 expr(Xs0,Xs1),   call(expr/2,2),        call(expr1/1,0),       [ expr1(Xs0),
                  put_y_unsafe_value(0,0),
                  put_y_value(1,1),
                  deallocate,            deallocate,
 expr(Xs1,Xs).    execute(expr/2)].      execute(expr1/1)].       rest_expr1 ].
```

Acknowledgements. This work was done within INTAS-93-1702.

Conclusion

We have presented a program transformation that uses Continuation Prolog as an intermediary language. The resulting programs can be translated further on to both binary Prolog and WAM-code. They can therefore be used for systems with and without environment stacks.

References

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

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

[3] Andreas Krall and Ulrich Neumerkel. The Vienna Abstract Machine. In Deransart and Małuszyński, pages 121–135.

[4] 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.

[5] 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.

[6] U. Neumerkel. A Transformation Based on the Equality between Terms. *Logic Program Synthesis and Transformation, LOPSTR 1993* Springer-Verlag, 1993.

[7] 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.

[8] P. v. Roy. Can Logic Programming Execute As Efficiently As Imperative Programming? Dissertation, UC Berkeley, December 1990.

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

[10] 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.

[11] 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.

[12] A. Taylor. High-Performance Prolog Implementation. Ph.D. dissertation, University of Sydney, June 1991.

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

[14] Zhou, Neng-Fa. On the Scheme of Passing Arguments in Stack Frames for Prolog. *11t ICLP94.*