Interprocedural register allocation for the WAM based on source-to-source transformations

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

Abstract

An approach for interprocedural register allocation for the WAM is presented which is based on source-to-source transformations of an intermediary language called Continuation Prolog. Continuation Prolog fills the conceptual gap between Prolog source code and the underlying abstract machine. Our approach does not require an analysis of the whole program. Only the definition of a predicate must be analyzed, but not its use. For certain kinds of predicates like DCGs no analysis is required at all. An implementation of our approach has been integrated into the WAM code generation of SICStus-Prolog. Our approach yields speedups up to 30% for existing benchmarks.

1 Introduction

Current Prolog implementations still lack many optimizations even for common applications like syntax analysis. Nonterminal e//0 illustrates the problem.

e $\longrightarrow$ "t".
e $\longrightarrow$ "o", e, e.

In a procedural language this is implemented with recursive descent. The input string is a global variable which is updated destructively. In Prolog the variable representing the input string is simulated with several different logic variables since logic variables only represent a single state.

$$e(\ [0't|\overset{Xs}{\overrightarrow{Xs],Xs}}).$$
$$e(\ [0'o|\overset{Xs0}{\overrightarrow{Xs0],Xs) \leftarrow e(Xs}}0,\overset{Xs1}{\overrightarrow{Xs1),e(Xs}}1,Xs).$$

These overheads can be observed in all Prolog machines. In most 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. While current optimizing Prolog compilers reduce these overheads, they do not remove them completely.

In previous work we have developed program transformations for Prolog [5] that are able to express certain optimizations which are not possible in the framework of fold/unfold [11]. Beside other applications, EBC-transformations allow to express comparatively low level optimizations on a source-to-source basis. Originally our transformations dealt with binary logic programs [12] therefore restricting their scope of application to binary Prolog systems. Recently our transformations were reformulated using a new intermediary language called Continuation Prolog [6] (which can be seen as a special case of Continuation Logic Programming [13]). With the help of Continuation Prolog programs are optimized to serve both BinWAM- and WAM-based systems. This paper focuses on WAM-based systems and presents a new way for compiling Continuation Prolog programs into WAM-code.

Limitations of fold/unfold. Traditional source-to-source transformations are severely limited for many applications by using plain Prolog. Every Prolog goal is seen independently of other goals. The context between goals cannot be expressed adequately. Either two consecutive goals are merged together completely (using definition and folding à la [7]) or they remain separated. Quite often, as in the case of list differences there are circular dependencies like in e//0 that are not resolvable by fold/unfold. For this reason we use a variant of Prolog —Continuation Prolog— that is able to express context between goals with ease.

Continuation Prolog. The meaning of a Continuation Prolog program is given by contint/1. Continuation Prolog rules are represented by facts $\twoheadleftarrow$/2. 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.

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

The relation between Continuation Prolog and Binary Prolog is straightforward. Each element C of a continuation Cs in Continuation Prolog corresponds to a term BinCont in Binary Prolog. A translation into WAM-code is given in subsequent chapters.

```
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).
```

Overview. Our approach to interprocedural register allocation is summarized by the following predicate.

```
contprolog_llprologallocated_(CProg0, LLProg, Anns) ←
    contprolog_(CProg0, Anns), % heuristics
    contprolog_ebctransformed_(CProg0,CProg, Anns), % Sect.2
    contprolog_llprolog_(CProg, LLProg, Anns). % Sect.3
```

In Sect.2 we describe how a Continuation Prolog program (CProg0) is first transformed with EBC-transformations. Sect.3 describes the translation from a Continuation Prolog program into a low-level Prolog program. We note that not all results of EBC-transformations can be translated into ll-Prolog. In order to reduce the nondeterminate search caused by contprolog_ebctransformed_/3 heuristics may be used (contprolog_/2). This paper deals only with EBC-transformations and the translation into low-level Prolog. Low-level Prolog is translated into SICStus' WAM instructions which may then be translated into emulator code or machine code [3].

Sect.4 presents a shortcut for DCGs. Sect.5 evaluates performance. Our approach is compared to related work in Sect.6.

2 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.

2.1 Equations of continuations

EBC-transformations introduce new alternative representations of continuations via equations. To some extent these equations correspond to definitions in fold/unfold. Equations are of the form $s \doteq t$. Both s and t are continuations (lists of continuation elements). With this equation every continuation c where $c = u \cdot s\theta \cdot v$ is equivalent to $d = u \cdot t\theta \cdot v$. For example, the equation [e(Xs0,Xs)] $\doteq$ [**e1**(Xs0), **r**(Xs)]. states that the new functors e1/1 and r/1 may serve as substitutes for e/2. The continuation [e(Ys0,Ys1), e(Ys1, Ys)] may now be replaced by [e1(Ys0), r(Ys1), e1(Ys1), r(Ys)]. In this paper we consider equations of the form

[old$(a_1, ...a_n, b_1, ...b_m)$] $\doteq$ [new1$(a_1, ...a_n)$, new2$(b_1, ...b_m)$]

where all a_i and b_i are distinct variables. These equations will split split up a continuation into two separate parts. The second part is intended for a predicate's output arguments.

2.2 Compilation of equations

Equality over continuation is implemented with syntactic unification only. Prior to execution, the equations are compiled into the program.

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 alternate representations of a term. In general all possible rewritings of the heads have to be added to the program. Since we restrict ourselves to equations of a very simple form, the following shortcut is possible instead.

All heads are rewritten. For each equation

[old$(a_1, ...a_n, b_1, ...b_m)$] $\doteq$ [new1$(a_1, ...a_n)$, new2$(b_1, ...b_m)$]

the clause [old$(a_1, ...a_n, b_1, ...b_m)$] $\twoheadleftarrow$ [new1$(a_1, ...a_n)$, new2$(b_1, ...b_m)$]

is added to the program. These clauses serve therefore as an interface from the old representation to the new one.

2.3 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. For the equations in this paper the following simplification rule suffices.

A clause $h \cdot r \twoheadleftarrow b \cdot r$ can be simplified to $h \twoheadleftarrow b$ if r is a single continuation element with all arguments being distinct variables and the variables in r do not occur in h and b.

2.4 Example

The translation of arithmetical expressions (taken from [9], Fig. 3.2) will serve as an example.

```
expr(I,T) -->                 expr(A+B,T) -->
    {integer(I)},                 expr(A,TA),
    !,                            expr(B,TB),
    {I = T}.                      ['add'(TA,TB,T)].
```

Translation into Continuation Prolog. Every goal is mapped into a single continuation element. Cuts do not need to be transformed into labeled cuts because the scope for the cuts will not change during this particular strategy.

```
[expr(I,T,Xs0,Xs)] <<-          [expr(A+B,T,Xs0,Xs)] <<-
  [ integer(I),                   [ expr(A,TA,Xs0,Xs1),
    !,                              expr(B,TB,Xs1,Xs2),
    I = T,                          Xs2 = ['add'(TA,TB,T)|Xs]
    Xs0 = Xs                      ].
  ].
```

Regrouping of output unifications. Output unifications are grouped into a single goal to simplify the splitting of output arguments. Instead of defining a separate predicate for each such case, we use the goal =/2 instead.

```
[expr(I,T,Xs0,Xs)] <<-          [expr(A+B,T,Xs0,Xs)] <<-
  [ integer(I),                   [ expr(A,TA,Xs0,Xs1),
    !,                              expr(B,TB,Xs1,Xs2),
    [I,Xs0] = [T,Xs]                Xs2 = ['add'(TA,TB,Tx)|Xs3],
  ].                                [Tx,Xs3] = [T,Xs]
                                  ].
```

Splitting output arguments and simplification. The second argument and the rest of the list difference are split into a separate continuation element. Notice that any split of arguments is possible and correct. However, only some of them will allow a subsequent simplification and will eventually be translatable into WAM-code.

In both clauses the continuation r/2 is now redundant. It is simply read by the head and written back in the last goal. We therefore remove these redundant parts. Note that this simplification step is only possible if r/2 really contains output arguments. If r/2 would contain some terms in the head, no simplification would be possible.

[expr(E,T,Xs0,Xs)] ⩦ [expr1(E,Xs0), r(T,Xs)].
[[Tx,Xs0] = [T,Xs]] ⩦ [=([Tx,Xs0]), r(T,Xs)].

[expr(E,T,Xs0,Xs)] ↞
  [expr1(E,Xs0),
    r(T,Xs)
  ].

[=([Tx,Xs0]), r(T,Xs)] ↞
  [[Tx,Xs0] = [T,Xs]].

[expr1(I,Xs0),~~r(T,Xs)~~] ↞
  [integer(I),
    !,
    =([I,Xs0]),
    ~~r(T,Xs)~~
  ].

[expr1(A+B,Xs0), ~~r(T,Xs)~~] ↞
  [expr1(A,Xs0),
    r(TA,Xs1),
    expr1(B,Xs1),
    r(TB,Xs2),
    Xs2 = ['add'(TA,TB,Tx)|Xs3],
    =([Tx,Xs3]),
    ~~r(T,Xs)~~
  ].

The predicate is now ready for direct compilation into WAM-code. If the generation of a binary Prolog program is desired, further transformation steps are required [6].

[expr1(I,Xs0)] ↞
  [integer(I),
    !,
    =([I,Xs0])
  ].

[expr1(A+B,Xs0)] ↞
  [expr1(A,Xs0),
    r(TA,Xs1),
    expr1(B,Xs1),
    r(TB,Xs2),
    Xs2 = ['add'(TA,TB,Tx)|Xs3],
    =([Tx,Xs3])
  ].

3 Compiling Continuation Prolog into WAM-code

Subset. Since the WAM uses a simple stacking regime for determinate control flow, only a subset of Continuation Prolog can be directly translated into WAM-code. The following Continuation Prolog programs are directly translatable into WAM-code without using the heap for AND-control.

All functors of continuation elements are classified according to their occurrence in the head.

1. Predicate functors. They only occur as the first element in the head.

2. Continuation functors. They occur only as the second element.

If a clause contains a continuation functor in its head, the body of the clause must be empty. The continuation functor must be the same for all clauses of the same predicate. For other clauses there is no restriction.

Low level built-ins. To keep our implementation simple by reusing the WAM-code generator of SICStus (plwam), the translation to WAM-code is not performed directly on Continuation Prolog programs. Instead, Continuation Prolog programs are translated back into regular Prolog extended with two new low-level built-ins. Both built-ins are of arbitrary arity.

'SINK'/n : ensures that all its arguments are located in the corresponding argument registers. It must only occur as the last goal of a clause. Code generation for 'SINK'/n is very similar to code generated for a regular goal with LCO.

When there are no other regular goals, 'SINK'/n is translated into a proceed instruction. Otherwise execute 'SINK'/n is generated. In this manner also blocked goals are executed without affecting the n valid registers.

'SOURCE'/n : ensures that all its arguments are using the corresponding argument registers. It occurs in a clause only directly after a regular goal (i.e., after a call instruction). Code generation is similar to head arguments.

In an experimental extension to the SICStus-Prolog compiler these built-ins have been integrated. Both built-ins are translated into the existing WAM instruction set. It was not necessary to extend the WAM. While 'SINK'/n is very close to a regular goal, the 'SOURCE'/n built-in required a more complex register allocation in the body of a clause.

The argument list of both built-ins consists of a list of decorated terms. A dash means that the argument is undefined. A term decorated with the structure +(X) means that the argument should be the value X. The following predicate pop/1 ensures that the second argument will be set to the integer 8250 while the value of the first argument is arbitrary. When using pop/1 only the second argument (or none) may be used with 'SOURCE'/2. Our transformation ensures the correct usage of these built-ins.

```
pop(china) ←
    'SINK'(-,+(8250)).

pop(C,P) ←
    pop(C),
    'SOURCE'(-,+P).

pop_void(C) ←
    pop(C),
    'SOURCE'(-,-). % P not needed.
```

Continuation Prolog to low-level Prolog. The translation of the Continuation Prolog subset to low-level Prolog is straight-forward. Predicate functors are mapped into heads and goals. Each argument of a continuation functor is mapped onto a different argument register. Continuation functors in the head are mapped onto 'SINK'/n goals, while continuation functors in the body are mapped onto corresponding 'SOURCE'/n instructions. As an optimization, output unifications (=/1) at the end of a rule are translated directly into 'SINK'/n goals.

```
[expr1(I,Xs0)] ↞                  expr1(I,Xs0) ←
  [  integer(I),                     integer(I),
     !,                              !,
     =([I,Xs0])                      'SINK'(+I,+Xs0).
  ].
[expr1(A+B,Xs0)] ↞                expr1(A+B,Xs0) ←
  [  expr1(A,Xs0),                   expr1(A,Xs0),
     r(TA,Xs1),                      'SOURCE'(+TA,+Xs1),
     expr1(B,Xs1),                   expr1(B,Xs1),
     r(TB,Xs2),                      'SOURCE'(+TB,+Xs2),
     Xs2 = ['add'(TA,TB,T)|Xs],      Xs2 = ['add'(TA,TB,T)|Xs],
     =([T,Xs])                       'SINK'(+T,+Xs).
  ].
```

Final example code. The effect of our transformation is summarized below. For the simple rule, all superfluous instructions have been removed. Only built-in calls remain. Also the recursive rule is improved significantly. Only two instruction for moving values remain (marked with %%). All other instructions are required because they read/create terms. The final program is 30% faster (cf. Sect.5).

```
% Prolog code           % Original WAM              % Optimized WAM             % ll-Prolog
expr(I, T, Xs0,Xs) ←                                                            expr(I, Xs0) ←
  integer(I),           [builtin(integer(0),else)   [builtin(integer(0),else)     integer(I),
  !,                    ,cutb                       ,cutb                         !,
  I = T,                ,get_x_val(1,0)             ,proceed].                    'SINK'(+I, +Xs0).
  Xs0 = Xs.             ,get_x_val(2,3)
                        ,proceed].
```

```
% Prolog code          % Original WAM              % Optimized WAM              % ll-Prolog
expr(A+B,              [get_str_x0((+)/2)          [get_str_x0((+)/2)           expr1(A+B, Xs0) ←
                       ,allocate                   ,unify_x_var(0)
 T, Xs0,               ,get_y_var(4,3)             ,allocate
 Xs) ←                 ,get_y_var(3,1)             ,unify_y_var(1)
                       ,unify_x_var(0)
                       ,unify_y_var(5)
                       ,put_y_var(1,1)
                       ,put_y_var(6,3)
                       ,init([0,2])                ,init([0])
 expr(A,TA,Xs0,Xs1),   ,call(expr/4,7)             ,call(expr1/2,2)               expr1(A, Xs0),
                       ,put_y_val(5,0)             ,get_y_first_val(0,0)%%        'SOURCE'(+TA,+Xs1),
                       ,put_y_first_val(2,1)
                       ,put_y_unsafe_val(6,2)
                       ,put_y_first_val(0,3)       ,put_y_val(1,0)%%
 expr(B,TB,Xs1,Xs2),   ,call(expr/4,5)             ,call(expr1/2,1)               expr1(B, Xs1),
 Xs2 = [                ,put_y_val(0,0)
                       ,get_list(0)                ,get_list(1)                   'SOURCE'(+TB,+[
                       ,unify_temp_var(0)          ,unify_temp_var(2)
                       ,unify_y_local_val(4)       ,unify_x_var(1)
   'add'(              ,get_temp_str('add'/3,0)    ,get_temp_str('add'/3,2)         'add'(
     TA,               ,unify_y_local_val(1)       ,unify_y_local_val(0)              TA,
     TB,               ,unify_y_local_val(2)       ,unify_x_local_val(0)              TB,
     T) | Xs].         ,unify_y_local_val(3)       ,unify_x_var(0)                    T) | Xs]),
                       ,deallocate                 ,deallocate
                       ,execute(true/0)].          ,execute('SINK'/2)].           'SINK'(+T,+Xs).
```

Blocked goals. SICStus-Prolog blocked goals are executed at call, execute and heapmargin_call instructions. At these points of execution allocation is aware of all life registers. Therefore, no special precaution must be taken in these cases. When a clause uses inline-built-ins at the end of a clause, SICStus generates an execute true/0 instruction instead of proceed. In our optimized clauses a call to a dummy predicate with higher arity is generated instead ('SINK'/n).

Block declarations for optimized predicates are currently not implemented. They can be simulated as follows. A test for the blocking condition is added to the optimized predicate/2. In case of it being true the auxiliary predicate blockable_predicate/4 is called. The output arguments of the suspended goal are passed back into the registers where they are expected to be by the callers of predicate/2. In this manner both the calling conventions for blocked goals and our optimized predicates can coexist.

```
% Simulating ← block predicate(-, ...).   ← block blockable_predicate(-, ?, ?, ?).
predicate(I1, ...) ←                      blockable_predicate(I1,..., O1,...) ←
    var(I1),                                  predicate(I1,...),
    !,                                        'SOURCE'(+O1,...).
    blockable_predicate(I1,..., O1,...),
    'SINK'(+O1,...).
predicate(_I1, ...) ←
    ... .
```

Limitations of EBC-register allocation. Many existing Prolog programs are written in a style that limits the application of our optimization. Mostly this is due to the improper usage of cut and output unifications. The nonterminal expr//2 (from [9], Fig. 3.2) illustrates the problem.

```
expr((X is Expr), Code) ←
   phrase(expr(Expr, X), Code).

expr(V, V) ⟶ {var(V)}, !.
expr(I, I) ⟶ {integer(I)}, !.
expr(A+B, T) ⟶ expr(A, TA), expr(B, TB), ['add'(TA,TB,T)].
expr(A-B, T) ⟶ expr(A, TA), expr(B, TB), ['sub'(TA,TB,T)].
expr(A*B, T) ⟶ expr(A, TA), expr(B, TB), ['mul'(TA,TB,T)].
expr(A/B, T) ⟶ expr(A, TA), expr(B, TB), ['div'(TA,TB,T)].
```

In expr//2 the second argument cannot be identified as an output argument because of the cut occuring after the last occurrence of the variable of the second argument. By placing the output unification after the cut the second argument becomes an output argument.

```
expr(V, T) ⟶ {var(V)}, !, {V = T}.
expr(I, T) ⟶ {integer(I)}, !, {I = T}.
```

4 Direct compilation of DCGs

Commonly DCGs are implemented by a preprocessing phase which maps each grammar rule onto a predicate. The string or list the DCG describes is represented with a list difference. All of these differences can be identified with our transformation based analysis. The preprocessing, transformation and register allocation phases can be bypassed by translating a grammar rule directly into our intermediary language ready for WAM-code generation.

Each nonterminal of arity n is translated into a predicate of arity $n+1$. The string is allocated in the second argument to avoid interference with first argument indexing. As an exception, for nonterminals of arity zero the input string is put into the first argument. The output argument is always allocated in the second argument ('SINK'(-,+Xs)).

```
a ⟶                        a(Xs) ←
   [].                        'SINK'(-,+Xs).

b(1) ⟶                     b(1, Xs) ←
   [].                        'SINK'(-,+Xs).

c(1,2) ⟶                   c(1, Xs, 2) ←
   [].                        'SINK'(-,+Xs).
```

```
qsorted([E|Es]) ⟶
    {partition(Es,E,Es1,Es2)},
    qsorted(Es1),
    [E],
    qsorted(Es2).
qsorted([]) ⟶
    [].
```

```
qsorted([E|Es], Xs0) ←
    partition(Es,E,Es1,Es2),
    qsorted(Es1, Xs0),
    'SOURCE'(-,+[E|Xs1]),
    qsorted(Es2, Xs1).
qsorted([], Xs0) ←
    'SINK'(-,+Xs0).

phrase_qsorted(Es, Xs0,Xs) ←
    qsorted(Es, Xs0),
    'SOURCE'(-,+Xs).
```

5 Benchmark results

Our transformation is particularly well suited to optimize differences. As Taylor remarks [15] "they are not prevalent in benchmark programs but skilled Prolog programmers make significant use of them." We present therefore both artificial predicates which show the potential of our optimization for differences as well as those of the existing benchmarks, where our optimization was applicable.

The transformations we applied are all mono-variant (each program point is translated into a single corresponding program point). Therefore no code duplication has taken place. In all of the examples the actual number of WAM instructions has been reduced. Measurements were made under SICStus 3#3 on a i486 DX2-66 with 16MB RAM.

For each benchmark, up to four versions are measured. "orig": the original predicate; "modif": a manually rewritten version to ensure steadfastness. In most of the cases these modifications were necessary to make EBC-transformations applicable. "EBC": using our optimizations; either via transformations or via direct DCG-translations. "tuned": further improvements that go beyond the optimizations in this paper. The speedup factors compare our programs with the version "modif" (or "orig" where no modification was necessary).

e and **de** are artificial programs to estimate the cost of handling differences. In addition to the original list difference up to three additional differences were added which are passed around without any modification. e generates all possible sentences of length 19. de uses a version of e//0 to parse a given sentence avoiding shallow backtracking.

qsort is a benchmark by D.H.D. Warren sorting a list of 50 given integers. The sort predicate uses list differences for the description of the sorted list. The differences are used in a rather unnatural manner: the end of the list is computed first and then the beginning. The middle element can thus be created with a put_list instead of a get_list instruction. In this manner no trail testing is necessary in the body. But qsort/2 as it is cannot benefit from our improved WAM code generation technique.

It seems to be more natural, however, to view sorting as describing the structure of a sorted list. The nonterminal qsorted//1 (see above) describes a sorted list. As a further improvement (tuned version) partition/4 was rewritten as a grammar rule. In this manner the list to be sorted always remains located in its dedicated register. All versions use a steadfast variant of partition/4 which does not create any choice points.

		orig.[ms]	modif.	EBC	tuned
expr	$\times 10^4$	2075	1.53	1.20	1.30
fact	10×10^3	156	-	1.06	2.05
fact	100×10^3	3296	-	1.02	1.05
qsort	$\times 10^3$	2502	1.00	1.02	1.14
query	$\times 10^2$	3208	-	1.11	1.21
serial	$\times 10^3$	3346	-	1.00	1.16
te	19×10^1	30342	-	1.20	-
e	19×10^1	15722	-	1.09	-
e+1	19×10^1	26424	-	1.55	-
e+2	19×10^1	35130	-	1.99	-
e+3	19×10^1	44294	-	2.29	-
de	19×10^5	9156	-	1.27	-
de+1	19×10^5	12150	-	1.67	-
de+2	19×10^5	15336	-	2.13	-
de+3	19×10^5	17346	-	2.41	-
d-divide	10×10^4	2587	1.18	-	1.03
d-log	10×10^4	905	1.55	-	1.00
d-ops	8×10^4	1769	1.25	-	1.05
d-times	10×10^4	2209	1.18	-	1.05
chat	$\times 10^1$	12980	-	1.23	1.31

expr was modified by moving output unification after cuts. The second output argument was optimized in the tuned version.
d-* are the differentiation benchmarks. They were modified by placing output unifications after cuts. While the EBC-strategy presented in this paper is not directly applicable another more advanced technique has been used to optimize one context argument.
serial: Only the predicate numbered/3 has been optimized. In the tuned version registers were passed over before/2.
query: The tables pop/2 and area/2 have been transformed. Further some output unifications were moved to allow the detection of another output argument
chat: The parser of chat80. The original version is the well known pre-expanded version. We wrote an alternate expansion for extraposition grammars similar to the DCG expansion. In the tuned version some rules which are not part of the extraposition grammar were integrated into the grammar.

6 Related Work

Comparison with mode analysis. Systems depending on mode analysis cannot obtain comparable results for the following reasons. In mode analysis "the mode of a predicate in a program indicates how its arguments will be instantiated when that predicate is called" [2]. Our approach is independent of the instantiations at calling time. Only the definition of a predicate (and the definitions of the predicates used herein) can influence the outcome of our analysis. Our approach is therefore more robust since separate or incremental analysis is possible. Furthermore in mode analysis the "modes are meaningful only when the control strategy has been specified" [2]. Our transformation optimizes the case of traditional Prolog control strategy but does not preclude different control strategies. Earlier we showed how block-declarations can be added to an already transformed program.

Aquarius Prolog. Van Roy's Aquarius Prolog [9] performs interprocedural register allocation with the help of *uninitialized register conversion* (Chapter 5.4.2). Arguments holding new unaliased variables at calling time are not initialized and are passed back in registers (uninit_reg) if last call optimization (LCO) can be preserved. However this optimization applies only to arguments which are always uninitialized variables. If they are initialized (as is the case for DCGs where the rest of the list difference is set to nil) no optimization can be observed. The overheads are 23%, 51%, and 86% for 1, 2, and 3 additional differences. Only by manually moving the unification with nil after the actual goal some optimizations can be observed. But even in this case an overhead of 3%, 5%, and 19% remains. We note that moving certain unifications after a goal may change the meaning of a program. A separate analysis is necessary to ensure that such a transformation is valid.

This contrasts clearly to EBC where optimizations are independent of the actual usage. In summary we can conclude that the benefits of integrating our approach into the highly optimizing Aquarius compiler would be higher speed and less sensitivity of the optimizer towards a predicate's use.

	additional differences			
	t[ms]	+1	+2	+3
init de	1434	1.23	1.51	1.86
uninit de	1433	1.03	1.05	1.19
init e	2808	1.27	1.58	1.88
uninit e	2160	1.01	1.02	1.05

Overheads for differences in Aquarius

Output value placement. Bigot, Gudeman and Debray propose to allocate output arguments in registers [1] and present a cost model for guiding allocation. In our setting many of the programs like p/2 cannot be optimized directly. As Debray remarks "last call optimization (LCO) often interferes with interprocedural register allocation". Most of these cases do not occur in the context of our transformation, because EBC cannot detect these opportunities directly.

← mode p(out,out).
p(X,Y) ← q(X,Y).
p(X,Y) ← q(Y,X).

However, by making output unification in the second clause more explicit, our transformations could be applied, provided that both arguments of q/2 are actually output arguments. Only then (when LCO is already lost), EBC is able to optimize the program.

p(X,Y) ← q(X,Y).
p(X,Y) ← q(YOut,XOut), X = XOut, Y = YOut.

Destructive updates. Tarau, Dahl, and Fall implement DCGs as HAGs (Hidden Accumulator Grammars) using backtrackable (on forward recursion destructive) assignment [13]. They extended the underlying abstract machine with new instructions whereas our approach does not require any modification on that level. It seems that HAGs make coroutining significantly more complex than our approach because in addition to registers *all* hidden states must be taken into account. One potential merit of hidden states is that they are trailed selectively instead of requiring space in each choice point. On the other hand each update of a hidden state now requires extra trail checking (and maybe also time stamping). Our approach produces the same effect if the underlying WAM uses register trailing.

Conclusion

We have presented a program transformation which can be used to perform interprocedural register allocations on both the WAM and the BinWAM. In future work we plan to support formalisms that go beyond simple DCGs like EDCGs [8] and develop guiding heuristics to direct the allocation phase.

Acknowledgments

This work was done within INTAS-93-1702.

References

[1] P. A. Bigot, D. Gudeman, S. K. Debray. Output Value Placement in Moded Logic Programs. 175–189 *ICLP 1994*, MIT Press.

[2] Saumya K. Debray, David Scott Warren. Automatic Mode Inference for Prolog Programs. 78–88 *SLP 1986*, IEEE.

[3] R. C .Haygood. Native Code Compilation in SICStus Prolog. 190–204 *ICLP 1994*, MIT Press.

[4] U. Neumerkel. Specialization of Prolog programs with partially static goals and binarization, Dissertation. Inst. für Computersprachen, TU-Wien, 1992.

[5] U. Neumerkel. A Transformation Based on the Equality between Terms. *LOPSTR 1993*, Springer-Verlag.

[6] U. Neumerkel. Continuation Prolog: A new intermediary language for WAM and BinWAM code generation. ILPS 95 Workshop on Sequential Implementation Technologies for Logic Programming Languages, Portland December 1995.

[7] M. Proietti, A. Pettorossi. Unfolding-definition-folding in this order, for avoiding unnecessary variables in logic programs. *PLILP 1991*, *LNCS 528*, 347–358, Springer-Verlag.

[8] P. v. Roy. A Useful Extension to Prolog's Definite Clause Grammar Notation. ACM SIGPLAN Notices, Vol. 24, No. 11, 1989, 132–134.

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

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

[11] H. Tamaki, T. Sato. Unfold/fold transformation of logic programs. S.-Å. Tärnlund, (Ed.), *2nd Int-l Logic Prog. Conf*, 127–138, Uppsala, 1984.

[12] P. Tarau, M. Boyer. Elementary logic programs. *PLILP 1990*, *LNCS 456*, Springer-Verlag.

[13] P. Tarau, V. Dahl, A. Fall. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. ILPS 95 Workshop on Visions for the Future of Logic Programming, Portland December 1995.

[14] J. Andrews, V. Dahl, P. Tarau. Continuation logic programming: Theory and Practice. ILPS 95 Workshop on Operational and Denotational Semantics of Logic Programs, Portland December 1995.

[15] A. Taylor. High-Performance Prolog Implementation. Ph.D. diss., Univ. of Sydney, June 1991.