A Prolog Positive Supercompiler

Stephan Diehl

FB 14 - Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, GERMANY , diehl@cs.uni-sb.de

Proceedings of "Arbeitstagung Programmiersprachen", editor Herbert Kuchen, published as Arbeitsbericht No. 58 des Institutes für Wirtschaftsinformatik, Westfälische Wilhelms-Universität Münster, 1997

Abstract. Supercompilation is a method for program specialization. It has been developed in the context of functional languages. We present a positive supercompiler for Prolog. We give the basic algorithm used in APROPOS and explain the different phases involved in supercompilation of logic programs. Then the operation and efficiency of the method is demonstrated by means of examples. Finally we compare our work to related work in supercompilation and partial evaluation.

1 Introduction

Supercompilation[1] is a program specialization technique originally developed by Valentin F. Turchin (e.g. [Tur86]) for a language called Refal . Supercompilation is composed of four steps: driving, generalization, folding and post-unfolding. Whereas in the original supercompiler positive and negative information, i.e. equality and inequality of variables and values was propagated, we restrict propagation to the case of positive information gained by unification. In [Sør94, SGJ94] a positive supercompiler for a first-order functional language is presented. We use folding, unfolding and post-unfolding strategies similar to those used in Sørensen's supercompiler to specialize Pure Prolog programs. In [GS94] the authors establish the correspondence between partial deduction [Kom92, Gal93, LS91], i.e. unfolding of predicate calls, and driving in positive supercompilation. But as we will demonstrate here, also the remaining steps of positive supercompilation (folding, post-unfolding and generalization) can be adapted to specialize logic programs.

2 APROPOS

Briefly, APROPOS builds the SLD-tree with partially known data. For every node it defines a new predicate. If it encounters a node, which is 'similar' to an existing one, it lets the node point to the existing one and stops building that branch of the tree. First we consider pure Prolog with disjunctions and conditionals:

$$\boxed{\begin{array}{lll} P & ::= C^* & v \ variable \\ C & ::= L :- G & c \ constructor \\ G & ::= L \mid (L,L) \mid (L;L) \mid (L- > L;L) & p \ predicate \ name \\ L & ::= p \mid p(X,\ldots,X) & \\ X & ::= c \mid v \mid c(X,\ldots,X) & \end{array}}$$

[1] short for supervised compilation: run a program, observe its behaviour and generate a new, more efficient program.

Motivated by the presentation of the algorithm used in LogiMix [JGS93] we use inference rules to define a relation $\theta, \Delta, G \rightarrow \theta', \Delta', G'$, which maps a goal G and the database Δ containing the source program to the specialized goal G' and the database Δ' containing the definitions for the specialized goal G' and all predicates specialized during specialization of G.

We write $\xrightarrow{l_i}$ to name the different relations $l_i \in \{$ *special*, *head*, *body*, *DB*, *def*, *mod*, *single*$\}$ defined by inference rules. For better readability we will write

$$\frac{G_1 \xrightarrow{l_1} G_1' \ \ldots\ G_n \xrightarrow{l_n} G_n'}{G_0 \xrightarrow{l_x} G_0'}$$

as a shorthand notation for $\dfrac{\Theta,\Delta,G_1 \xrightarrow{l_1} \Theta_1,\Delta_1,G_1' \ \ldots\ \Theta_{n-1},\Delta_{n-1},G_n \xrightarrow{l_n} \Theta_n,\Delta_n,G_n'}{\Theta,\Delta,G_0 \xrightarrow{l_0} \Theta_n,\Delta_n,G_0'}$

Θ denotes a substitution in the logic programming sense. In addition we assume, that there is a global database Δ for entries of the form $dbp(\_,\_,\_)$, $dbc(\_,\_,\_,\_,\_)$, $db(\_)$ and $src(\_,\_)$. Initially for every clause H :− B of the source program, we have an entry $src(H, B)$ in the database. Furthermore it contains the entry $db(0)$. We give the basic algorithm used in APROPOS as a set of inference rules in figures 1, 5 and 2.

For some cases marked with $(*)$ we only give general rules. In APROPOS we used more specific rules to obtain simplified residuals. Furthermore we ignore in the following formal specification the details, which would be necessary to describe forward unification as it was implemented in APROPOS . Basically after partially evaluating a subterm we replace all variables in that subterm by ground terms, e.g. `gvar(4)`. As a result the bindings of unification are only propagated forward. As is well known, backward unification would interfere with nonlogical predicates like `var(X)` and with disjunctions and conditionals.

Finally the notation $G \Rightarrow G'$ states that G evaluates to G'. In the special case $H \sqcup G \Rightarrow U$ we state that the unification of H and G yields U, in other words the most general unifier $mgu(H, G) = \sigma$ exists. Note, that the bindings are propagated, i.e. $G \Rightarrow G'$ is short for $\theta, \Delta, G \Rightarrow \theta', \Delta, G'$ where all bindings which take place during evaluation of G are recorded in θ', in the special case of unification we have $\theta' = \sigma \circ \theta$.

Each occurrence of _ denotes a fresh anonymous variable. By $A \sqsubseteq B$ we mean that B is less general than A, i.e. $A \sqcup B \Rightarrow B$.

2.1 Database Entries

Assume we specialized a program with respect to the goal $G = p(X_1, \ldots, X_n)$. Furthermore assume, that it was declared in the meta-knowledge, that p/n should be specialized, then the database will contain an entry $dbp(1, G, S)$ where $S = p(Y_1, \ldots, Y_m)$ and $Y_1, \ldots, Y_m$ are the variables occurring in $X_1, \ldots, X_n$. Thus we have $p(X_1, \ldots, X_n) \rightarrow$ $call\_db(1, p(Y_1, \ldots, Y_m))$ and similar entries for all predicates specialized during the specialization of G. The predicates in the database are then post-unfolded (see section 2.5) and the resulting database entries are translated into program clauses as shown in figure 3. [2]

The database contains definitions for the specialized predicates, e.g. for the above goal $call\_db(1, p(Y_1, \ldots, Y_m))$ there are entries of the form $dbc(1, K, H, B, S')$ repre-

[2] We only give the two important translation rules here, the other rules just traverse the body of a clause.

Fig. 1. Specialization Rules ($\to$)

G $\to$ G if G is atomic

Use residual of previously specialized similar goal.

$$\left\{\begin{array}{l} \dfrac{G \xrightarrow{\text{head}} G'}{G \to G'} \text{ (folding)} \\[2ex] \dfrac{G \xrightarrow{\text{body}} G'}{G \to G'} \text{ (folding)} \end{array}\right.$$

Similar rules are used by most partial evaluators for Prolog

$$\left\{\begin{array}{ll} \dfrac{G_1 \to G_1' \qquad G_2 \to G_2'}{(G_1,G_2) \to (G_1',G_2')} & (*) \\[2ex] \dfrac{G_1 \to true \qquad G_2 \to G_2'}{(G_1 -> G_2 ; G_3) \to G_2'} & \dfrac{G_1 \to fail \qquad G_3 \to G_3'}{(G_1 -> G_2 ; G_3) \to G_3'} \\[2ex] \dfrac{G_1 \to G_1' \qquad G_2 \to G_2' \qquad G_3 \to G_3'}{(G_1 -> G_2 ; G_3) \to (G_1' -> G_2' ; G_3')} & \\[2ex] \dfrac{G_1 \to G_1' \qquad G_2 \to G_2'}{(G_1 ; G_2) \to (G_1' ; G_2')} & (*) \\[2ex] \dfrac{\Theta, G \Rightarrow \Theta_1, true \ \ \ldots \ \ \Theta, G \Rightarrow \Theta_n, true}{\Theta, G \to \Theta, (\delta_1 ; \ldots ; \delta_n)} & \text{if } G \text{ is evaluable (all solutions)} \\ \multicolumn{2}{l}{\text{where } \delta_i \text{ is a conjunction of unifications, such that } \Theta, \delta_i \Rightarrow \Theta_i} \\[2ex] \dfrac{G \Rightarrow true}{G \to true} & \text{if } G \text{ is evaluable (first solution)} \\[2ex] \dfrac{G \Rightarrow fail}{G \to fail} & \text{if } G \text{ is evaluable} \end{array}\right.$$

The goal is unfolded and then specialized.

$$\left\{ \dfrac{G \xrightarrow{\text{special}} G'}{G \to G'} \text{ if } G \text{ should be specialized} \right.$$

G $\to$ G otherwise

Fig. 2. Database Access Rules($\xrightarrow{\text{DB}}$, $\xrightarrow{\text{mod}}$)

Database Lookup

$$\left\{\begin{array}{l} \dfrac{E \in \Delta \qquad E \sqcup H \Rightarrow H'}{H \xrightarrow{\text{DB}} H'} \\ \text{where } E \in \Delta \text{ yields an element of } \Delta \text{ with new variables} \end{array}\right.$$

Modification of Database

$$\left\{ \Delta \xrightarrow{\text{mod}} \Delta' \text{ (overwrite } \Delta \text{ by } \Delta') \right.$$

Fig. 3. Converting database entries into program clauses

Generate a Prolog program from the database

$$\left\{\begin{array}{l} call\_db(N, p(Y_1, \ldots, Y_n)) \ \longmapsto \ \text{db\_N\_p}(Y_1, \ldots, Y_n) \\[2ex] \dfrac{B \longmapsto B'}{dbc(N,K,H,B,p(Y_1,\ldots,Y_n)) \ \longmapsto \ \text{db\_N\_p}(Y_1, \ldots, Y_n) \text{ :- } B'} \end{array}\right.$$

senting the K'th clause of the definition of the specialized predicate where $H \sqsubseteq G$ and $S' \sqsubseteq S$.

The database entry $dbp(N, H, S)$ states, that the predicate call H has been specialized to the call S with index N. Finally $db(N)$ is the number of currently defined specialized predicates.

In the next sections the different actions performed by APROPOS are discussed.

Fig. 4. Specialization Rules ($\xrightarrow{\text{special}}, \xrightarrow{\text{def}}, \xrightarrow{\text{simp}}$)

Lookup those clauses in the source program which match the current goal. Generate a call to a new predicate defined by the specialization of these clauses.

$$\left\{\begin{array}{l} \dfrac{\begin{array}{l} src(G,\_) \xrightarrow{\text{DB}} src(H_1,B_1) \\ \ldots\ src(G,\_) \xrightarrow{\text{DB}} src(H_k,B_k) \\ db(\_) \xrightarrow{\text{DB}} db(N) \quad G \xrightarrow{\text{simp}} S \quad \Delta \xrightarrow{\text{mod}} \Delta' \\ [H_1,B_1,G,S,N+1,1] \xrightarrow{\text{def}} [] \\ \ldots\ [H_k,B_k,G,S,N+1,k] \xrightarrow{\text{def}} [] \end{array}}{\text{G} \xrightarrow{\text{special}} \text{call\_db(N+1,S)}} \quad \text{(driving)} \\ \\ \text{where replacing } db(N) \text{ by } db(N+1) \\ \text{and adding } dbp(N+1,G,S) \text{ to } \Delta \text{ yields } \Delta' \end{array}\right.$$

Generate the K-th clause of the definition of a new predicate (with index N+1). The body of the clause is first specialized.

$$\left\{\begin{array}{l} \dfrac{\begin{array}{l} G \sqcup H \Rightarrow U \quad \Delta \xrightarrow{\text{mod}} \Delta' \\ B \rightarrow B' \quad \Delta'' \xrightarrow{\text{mod}} \Delta^* \end{array}}{\text{[H,B,G,S,M,K]} \xrightarrow{\text{def}} []} \quad \text{(driving)} \\ \\ \text{where adding of } dbc(M,K,H,pending,S) \text{ to } \Delta \\ \text{yields } \Delta' \text{ and replacing } dbc(M,K,H,pending,S) \\ \text{in } \Delta'' \text{ by } dbc(M,K,H,B',S) \text{ yields } \Delta^* \end{array}\right.$$

Generate a call pattern from a goal, i.e. the predicate name and the free variables in the goal.

$$\left\{\begin{array}{l} p(X_1,\ldots,X_k) \xrightarrow{\text{simp}} p(Y_1,\ldots,Y_n) \\ \\ \text{where } Y_1,\ldots,Y_n \text{ are the variables} \\ \text{occurring in } X_1,\ldots,X_k \end{array}\right.$$

2.2 Driving

When a predicate call is encountered which according to the meta-knowledge should be specialized, we compute the list of all matching[3] clauses $H_1 : -B_1, \ldots, H_k : -B_k$ and specialize the bodies of these clauses (c.f. $\xrightarrow{\text{special}}$). Now we define a new predicate which is defined by these specialized clauses. Finally the predicate call is replaced by a call to the newly created predicate. When a predicate call is encountered, which has been specialized before, it is also replaced by a call to the specialized predicate. This works in analogy to defining the specialized function f' as described above for functional languages.

[3] By matching we mean that the predicate call P is unified with the head H of a clause $H : -B$ and thus some of the variables in the body B get instantiated, more precisely: if

Fig. 5. Specialization Rules ($\xrightarrow{\text{head}}, \xrightarrow{\text{body}}$)

If the current goal is less general than the head of a previously specialized goal, then generate a call to that specialized goal.	$\left\{ \dfrac{\begin{array}{l} dbp(\_,\_,\_) \xrightarrow{\text{DB}} dbp(N,H,S) \\ G \sqsubseteq H \quad H \sqcup G \Rightarrow U \end{array}}{\text{G} \xrightarrow{\text{head}} \text{call\_db(N,S)}} \right.$ (generalization)
If the current goal is less general than the body of a clause defining a previously specialized goal, then generate a call to that specialized goal. The specialized goal must be defined by a single clause.	$\left\{ \dfrac{\begin{array}{l} dbc(\_,\_,\_,\_,\_) \xrightarrow{\text{DB}} dbc(N,K,H,B,S) \\ B \sqcup G \Rightarrow U \quad G \sqsubseteq B \\ dbc(N,K,H,\_,S) \xrightarrow{\text{DB}} dbc(N,K,H,B',S) \\ U \sqsubseteq B' \quad B' \sqsubseteq U \quad N,K \xrightarrow{\text{single}} N' \end{array}}{\text{G} \xrightarrow{\text{body}} \text{call\_db(N',S)}} \right.$

2.3 Folding

There are two cases of folding in APROPOS . One is the replacing of a predicate call by a call to a specialized predicate (c.f. $\xrightarrow{\text{head}}$). The other case is when we specialize an expression e and e matches the body b of a clause of a specialized predicate p (c.f. $\xrightarrow{\text{body}}$). In this case, the expression e is replaced by a call to that specialized predicate p. Actually, if p has more than one defining clause, a new specialized predicate p' is defined as $p' :- b$, the clause $p :- b$ is replaced by $p :- p'$ and e is replaced by a call to p' (c.f. $\xrightarrow{\text{single}}$).

Fig. 6. Basic Specialization Rules ($\xrightarrow{\text{single}}$)

If the predicate is not defined by a single clause ($K > 0$), then generate a new predicate defined by the body of the K-th clause and replace the body of the K-th clause by a call to the new predicate.	$\left\{ \begin{array}{l} N, 0 \xrightarrow{\text{single}} N \\ \\ \dfrac{db(\_) \xrightarrow{\text{DB}} db(M) \qquad \Delta \xrightarrow{\text{mod}} \Delta'}{N,K \xrightarrow{\text{single}} N'} \\ \\ \text{where replacing } dbc(N,K,H,B,S) \text{ by} \\ dbc(N,K,H,db(M+1,S),S),\ db(M) \\ \text{by } db(M+1) \text{ in } \Delta \text{ and adding} \\ dbc(M+1,0,H,B,S) \text{ to } \Delta \text{ yields } \Delta' \end{array} \right.$

2.4 Generalization

If we look for a suitable predicate p to replace the body b of a clause (c.f. $\xrightarrow{\text{body}}$), the body of p must be comparable and moreover more general than b. Furthermore after unifying these bodies we have to make sure, that no variables get bound, which are local to the body of p, i.e. do not occur in the head, e.g.:

$mgu(P,H) = \theta$ then the instantiated body is $\theta(B)$.

```
current goal:    q(8,9),r(9,Y)
defining clause: p(A,B) :- q(A,Z),r(Z,B).
```

In the above example, we can't replace the goal by `p(8,Y)`, because unifying the goal and the clause body yields `A=8,Z=9,B=Y`, i.e. the local variable is bound to 9. Unfolding the call `p(8,Y)` yields `q(8,Z),r(Z,Y)` and thus the restriction, that Z has to be 9, is lost. This generalization scheme can lead to overgeneralization, but it is the least restrictive rule, which preserves the semantics. By overgeneralization we mean, that instead of specializing predicate calls they will be folded using predicates, which have been specialized before but are much too general, i.e. this generalization scheme preserves the semantics but can disable possible specializations. To influence this behaviour, the user can specify additional restrictions in the meta-knowledge by defining the predicate $generalize/1$.

2.5 Post-Unfolding

In the body B of the specialized clause $dbc(N, K, H, B, S)$ predicate calls of the form *call_db*(M, X) may occur. Post-unfolding unfolds such calls in case there is only one call to that specialized predicate in all clauses in the database. After post-unfolding a predicate call the defining clauses of that predicate are removed from the database. We could also post-unfold $dbc(N, K, H, B, S)$ for $N > 1$ by constructing a disjunction of the bodies of each clause. Actually post-unfolding does not unfold all such calls because of recursion. Consider this simple example

```
p(X) :- X=[] -> true ; (X=[_|R],p(R)).
```

Unfolding `p(R)` would only complicate the definition of `p`, but we could not deleted the definition of the predicate `p`. APROPOS computes the strongly connected components of the call graph, removes all strongly connected components with more than one element and all trivial cycles, i.e. a predicate calling itself. The result is a directed acyclic graph and thus can be topologically sorted. Finally APROPOS unfolds those predicates only called once in reverse topological order and removes their definitions.

2.6 Meta-Knowledge

User-defined predicates (sometimes called filters) encoding meta-knowledge to control specialization have been used in several partial evaluators [FF88, LS90, Con90]. In APROPOS the user can control specialization, evaluation, generalization and post-unfolding. Those predicates can be specialized of which we have a definition, i.e. all predicates besides the system predicates. To reduce the number of newly created predicates, the user can specify, which predicates should be specialized by defining the predicate *should_specialize*$/1$, which given a predicate call decides whether to specialize it or not. By defining the predicate $evaluable/1$ the user can also control, which predicate calls are evaluated. Usually all predicate calls which are not side-effecting and have only ground input parameters can be evaluated. Unfortunately there is no clear distinction of input and output parameters in Prolog. Most calls with nonground parameters can also be evaluated. For example, we can evaluate `reverse([A,B,C],L)`, but we can't evaluate `reverse([F|R],L)`. So mode and groundness information is only a crude heuristic. In offline partial evaluators a preprocessing phase called 'binding-time analysis' annotates calls, so that at partial evaluation time the annotation determines whether a call is evaluated. In APROPOS we let the user specify 'heuristics', when to evaluate a predicate call. For the example above such a heuristic could be that a list can be reversed if it is finite:

```
evaluable(reverse(X,L)) :- nonvar(X),X=[].
evaluable(reverse(X,L)) :- nonvar(X),X=[A|R],evaluable(reverse(R,_)).
```

This strategy leads to a certain kind of binding-time improvement of the source-program. Since we cannot annotate program points in our system, we can introduce several versions of the same predicate but with different names. These predicate names encode some property of the program point they are called at and we can define specific rules, when to evaluate these calls. For the same reasons as in LogiMix, namely efficiency, we let the user specify, whether the specializer should consider all solutions to an evaluable predicate call or only the first one. In the latter case if the behaviour of the program depends on another solution of this predicate, APROPOS does not preserve the semantics. Note, that if the definitions of all predicates are accessible, APROPOS will also work without any partial evaluation of predicates. Thus it can do pure positive supercompilation, but the gains in efficiency are less.

2.7 Other Features and Full Prolog

Since we use forward unification and the user can specify conditions, when to evaluate goals, APROPOS can in an admittedly restricted way cope with side-effecting predicates like `assert` and `retract`. Another conservative strategy is, that APROPOS never unfolds a call, if the defining clause contains a cut.

In the case of specialization of disjunctions and conditionals, the evaluation of predicates with multiple solutions and the specialization of predicates with several defining clauses we compute and propagate the common bindings, e.g. after specializing the disjunction `(X=a(1,b(U)) ; X=a(Z,b(1)))` APROPOS will propagate the common binding `X=a(_,b(_))`. Similar methods have been used in FUSE [WCRS91] and Mixtus, in the latter it is called anti-unification.

3 Experiments

Pure positive supercompilation of the standard `append/3` predicate yields good results, e.g. for `append([1,2,3],X,Y)` we got the residual program:

```
db_1_append(A,[1|B]) :- B=[2|C], C=[3|D], D=A.
```

3.1 KMP Test

Another way to test the power of a program specializer is the KMP test (c.f. [CD89, SGJ94]). After specializing a general pattern matcher with respect to a fixed pattern we compare the residual program to the result of the Knuth-Morris-Pratt algorithm (c.f. [KMP77]) which constructs a deterministic finite automaton.

```
match(P,S) :- loop(P,S,P,S).
loop([],SS,OP,OS).
loop([P|PP],[S|SS],OP,OS) :- P=S -> loop(PP,SS,OP,OS) ; next(OP,OS).
next(OP,[]).
next(OP,[S|SS]) :- loop(OP,SS,OP,SS).
```

The above pattern matcher is not optimal. Each time a match fails (in the third clause of `loop`), the first character of the string is removed and the matching starts again (in the second clause of `next`). To simulate Sørensen's positive supercompiler we used the following meta-knowledge.

```
should_specialize(X).        % always unfold
evaluable1(X) :- ground(X).  % evaluate all ground calls, consider
                             % only the first solution
evaluable(X=Y) :- ground(X); ground(Y).  % evaluate unification if at
                                        % least one argument is ground

postunfold_single_clause_predicates_only :- true. % f-functions only
generalize(X) :- fail.                            % identical folding
```

Supercompiling the predicate `match(A,B)` with respect to `A=[a,a,b]` yields the following residual program.

```
db_1_match(A) :- db_2_loop(A).
db_2_loop([A|B]) :- A=a -> db_4_loop(B); db_3_next(A,B).
db_3_next(A,B) :- db_2_loop(B).
db_4_loop([A|B]) :- A=a -> db_7_loop(B) ; db_5_next(A,B).
db_5_next(A,B) :- A=a -> db_4_loop(B) ; db_3_next(A,B).
db_7_loop([A|B]) :- A=b -> true ;  A=a -> db_7_loop(B) ; db_5_next(A,B).
```

This residual program works like a deterministic finite automaton and is very close to the residual program obtained by positive supercompiling a general pattern matcher in a functional language (c.f. [SGJ94, Sør94]). The residual pattern matcher keeps track of the recognized prefix in its state (every clause encodes a state), i.e. it does not look at a character in the string more than once. Actually as in the functional case there is one redundant test in the above residual. In `db_5_next` the variable `A` is compared to `a` although the calls in `db_4_loop` and `db_7_loop` guarantee that `A` does not unify with `a`.

3.2 Specializing an Interpreters and Compiling Actions

We implemented an IMP interpreter based on the structural operational semantics given in [Win93]. Supercompiling the IMP interpreter with a given program for computing Fibonacci numbers and an unknown variable binding reduced the runtime by 20 percent. Similar speedups can be achieved with existing partial evaluators [4].

In [BP93] the authors describe how they compile actions by partial evaluation of an action interpreter written in Scheme. We did a similar experiment. We wrote a language prototyping system, which uses action semantics descriptions of a source language to convert source language programs into actions. These actions are executed by an action interpreter based on an structural operational semantics for action notation. Supercompiling the prototyping system with respect to a given source language description (mini-Δ in [BMW92]) and a simple source language program reduced the runtime by 65 percent, i.e. we got a speedup of 2.8. For more complicated source language programs, we couldn't achieve quite as good results. We expect, that binding time improvements in the action interpreter would improve the results.

[4] e.g. "Typical speed-ups for normal Prolog programs range from 3 to 20 %." [Sah90]

4 Related Work

Valentin F. Turchin's (e.g. [Tur86]) system was written in and for a functional language called Refal , which was first developed in 1968 and although being very innovative has never achieved great attention in the western world. In Turchin's supercompiler positive and negative information, i.e. equality and inequality of variables and values was propagated. In APROPOS positive information is propagated by unification (see section 2.2).

Sørensen's positive supercompiler was written for a first-order functional language tailored for supercompilation. In a functional language all arguments of a function are input parameters. In a logical language this is not the case. Mode information has to be computed or as in our system provided by the user. Furthermore in Sørensen's functional language there is only pattern matching for non-nested, linear patterns in the first argument of a function. In Prolog we have powerful pattern matching via unification on all arguments. Finally in Prolog we have backtracking, i.e. not a single-threaded store. On the other hand Sørensen proved, that his supercompiler preserves the semantics of the original program. Our implementation of supercompilation for logic programs applies transformations similar to the fold/unfold transformations presented in [TS84, PP93, PP94, BC93]. In that framework supercompilation can be regarded as a strategy which controls when to fold or unfold.

In partial evaluators like LogiMix calls are folded, which have been encountered before[5]. In contrast to APROPOS there is no generalization, no folding of bodies and no post-unfolding [6]. In partial evaluators most optimizations rely on the distinction of static and dynamic data. In offline partial evaluators the source program has to be annotated by a preprocessing phase called 'binding-time analysis'. In online systems evaluation of expressions is decided on the fly as it is done in APROPOS . As noted in [CD93] one can also combine online and offline methods to get the best of both.

We have also looked into self-application of APROPOS , but do not yet have any interesting results. As noted in [Jon94], one might need a preprocessing phase rather different from binding-time analysis. In order to achieve this, one has to find general, but powerful rules when to specialize or evaluate a goal.

5 Conclusion

We used techniques found in partial evaluators to implement a supercompiler for Prolog. We explained how it works and demonstrated its efficiency by means of examples. As expected APROPOS passes the KMP test. The experimental results suggest that APROPOS is as powerful as the more sophisticated partial evaluators for Prolog. We hope the existence of a supercompiler for Prolog - a widely used language compared to Refal - will lead to new and more interest in the pioneer work of Valentin Turchin.

References

[BC93] A. Bossi and N. Cocco. Basic transformation operations which preserve computer answer substitutions of logic programs. *Journal of Logic Programming*, 16(1, 2):47–87, 1993.

[5] A special case of this scheme is called loop detection in Mixtus.

[6] This is not quite true for Mixtus.

[BMW92] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an Action Semantics Directed Compiler Generator. In *CC'92, LNCS 641*. Springer Verlag, 1992.

[BP93] Anders Bondorf and Jens Palsberg. Compiling Actions by Partial Evaluation. *FPCA'93*, 1993.

[CD89] C. Consel and O. Danvy. Partial Evaluation of Pattern Matching in Strings. *Information Processing Letters*, 30(2), 1989.

[CD93] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *POPL'93*. 1993.

[Con90] Charles Consel. *The Schism Manual, Version 1.0.* Yale University, New Haven, Connecticut, December 1990.

[FF88] H. Fujita and K. Furukawa. A self-applicable partial evaluator and its use in incremental compilation. *New Generation Computing*, 6(2,3):91–118, 1988.

[Gal93] J. Gallagher. Tutorial on specialisation of logic programs. In *Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, June 1993*, pages 88–98. New York: ACM, 1993.

[GS94] R. Glück and M.H. Sørensen. Partial Deduction and Driving are Equivalent. In *PLILP'94*. 1994.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Englewood Cliffs, NJ: Prentice Hall, 1993.

[Jon94] N. D. Jones. The Essence of Program Transformation by Partial Evaluation and Driving. In *Logic, Language and Computation*, volume 792. Springer LNCS, 1994.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2), 1977.

[Kom92] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Meta-Programming in Logic, Uppsala, Sweden, June 1992 (Lecture Notes in Computer Science, vol. 649)*, pages 49–69. Berlin: Springer-Verlag, 1992.

[LS90] A. Lakhotia and L. Sterling. ProMiX: a Prolog Partial Evaluation System. In L. Sterling, editor, *The Practice of Prolog*. MIT Press, 1990.

[LS91] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.

[PP93] M. Proietti and A. Pettorossi. The loop absorption and the generalization strategies for the development of logic programs and partial deduction. *Journal of Logic Programming*, 16(1, 2):123–161, 1993.

[PP94] M. Proietti and A. Pettorossi. Completeness of Some Transformation Strategies for Avoiding Unnecessary Logical Variables. In *Proc. of the 11th International Conference on Logic Programming*. MIT Press, 1994.

[Sah90] Dan Sahlin. The Mixtus Approach to Automatic Partial Evaluation of Full Prolog. In *Proc. of the 1990 North American Conference on Logic Programming*. MIT Press, 1990.

[SGJ94] M.H. Sørensen, R. Glück, and N. D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation and GPC. In *ESOP'94*. Springer LNCS, 1994.

[Sør94] M.H. Sørensen. Turchin's Supercompiler Revisited: An Operational Theory of Positive Information Propagation. Master's thesis, Department of Computer Science, University of Copenhagen, 1994.

[TS84] H. Tamaki and T. Sato. Unfold/Fold Transformation of Logic Programs. In S. Tarnlund, editor, *Proc. of Second International Conference on Logic Programming*. 1984.

[Tur86] V.F. Turchin. The Concept of a Supercompiler. *ACM TOPLAS*, 8(3), 1986.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *FPCA'91*, volume LNCS 523, pages 165–191. Springer, 1991.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.

This article was processed using the LaTeX macro package with LLNCS style