Sequential Implementation of Parallel Narrowing

Daniela Genius*

Lehr- und Forschungsgebiet Informatik II
(Prof. Dr. Michael Hanus)
RWTH Aachen, D-52056 Germany

Abstract

Parallel narrowing has been proposed by Antoy, Echahed and Hanus. They generalize to narrowing the notion of necessary set of redexes introduced by Sekar and Ramakrishnan. We present a high-level implementation by transforming weakly orthogonal, constructor-based programs into Prolog with the help of parallel definitional trees. For strategies based on parallel rewriting, this is particularly difficult, because we have to take into consideration the whole term. A better strategy is obtained by combining parallel narrowing with simplification, exploiting both determinism and expression parallelism. We point out the advantages of the new strategy by comparative measurements.

1 Introduction

Functional logic languages combine the advantages of functional languages, such as higher order functions and efficient evaluation with those of logic languages like search, logical variables and dealing with partial information. A number of proposals have been made. A recent summary is found in [8]. A sound and complete semantics is *narrowing* [19]. Informally speaking, a narrowing step is an instantiation of goal variables followed by a reduction step on the instantiated goal.

Example 1.1 Consider the rules describing the addition and less-or-equal relation[1]:

$0 + Y \to Y$	*(A1)*	$0 \leq X \to true$	*(L1)*
$s(X) + Y \to s(X + Y)$	*(A2)*	$s(X) \leq 0 \to false$	*(L2)*
		$s(X) \leq s(Y) \to X \leq Y$	*(L3)*

An equation $X + s(0) \leq s(0) \approx true$ is narrowed to the trivial equation $true \approx true$ by instantiating X with 0 and applying rule *A1*. To apply *L3*, choose the identity substitution. On the resulting equation $0 \leq 0 \approx true$ *L1* can be applied, yielding the trivial equation $true \approx true$. □

Narrowing allows the selection of rule and position, and therefore has a huge search space. Functional logic languages with an eager narrowing semantics, such as ALF [7] and SLOG [5], select innermost positions for narrowing steps. Lazy narrowing strategies, on the other hand, generally apply steps at outermost positions, and consider inner positions only if they are demanded. The language BABEL also comprises lazy evaluation [17]. However, this strategy performs some variable instantiations that become unnecessary due to steps later in the derivation [3]. For the class of inductively sequential systems, *needed narrowing* [3] is a strategy which extends an approach of Huet and Lévi [12] to narrowing. Needed narrowing is optimal wrt. the number of solutions as well as the length of derivations. Our goal is to extend needed narrowing to a class which is more adequate for logic programming: *weakly orthogonal* systems allow overlapping left hand sides of rules.

Example 1.2 The disjunction is denoted by overlapping rules:

$$true \vee X \to true \qquad X \vee true \to true \qquad false \vee false \to false$$

Without look-ahead we do not know which argument of term $t_1 \vee t_2$ should be narrowed first. □

*genius@zeus.informatik.rwth-aachen.de

[1] Natural numbers are represented by terms built from 0 and s.

A conservative extension is *weakly needed narrowing* [2], which leaves the decision to nondeterministic choice. The classical approach until now was to try to detect determinism in order to reduce the possible choices. Loogen and Winkler [15] proposed a *dynamic cut*, cutting off alternative solutions at run time if no variables in the goal have been bound. However, dynamic cut is highly dependent on the order of rules and moreover, a Prolog implementation turns out to be much too complicated and inefficient [6]. The most promising approach until now has been *lazy narrowing with simplification* [9], where determinism is exploited by performing reduction steps 'as far as possible'. A new approach is the application of steps at multiple positions at once, so called *narrowing multisteps.*

As we have seen in example 1.2, we have to deal with *sets* of positions in the case of overlapping rules. Sekar and Ramakrishnan show how to minimize these sets [18]. They prove optimality among all rewrite strategies that do not consider the right hand sides of rules. This work is the background of [2], where the approach in extended to narrowing.

In the next section, we recall some basic notions of term rewriting that are necessary in our context and sum up previous related results. In section 3 we quote some of the results of [2] to explain the underlying ideas of our implementation and show translation schemes. Section 4 gives a short insight on run times and effects on the search tree between our parallel strategies and other approaches. We conclude by summing up some extensions and give an outlook. and give an outlook.

2 Preliminaries

We assume familiarity with the notions of term rewriting [4] and sum up some basic definitions. Consider a many-sorted *signature*, partitioned into the disjoint sets $\mathcal{C}$ of constructors and $\mathcal{F}$ of defined functions, $\mathcal{X}$ the set of variables. The sets of *expressions* and *terms* are denoted by $\mathcal{T}(\mathcal{C} \cup \mathcal{F}, \mathcal{X})$ and $\mathcal{T}(\mathcal{C}, \mathcal{X})$, respectively. $Var(t)$ denotes the set of all variables in t. A term t is *ground* if $Var(t) = \emptyset$. A term $f(t_1, \ldots, t_n)$ is a *pattern* if $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$. A *constructor-based term rewriting system* (*TRS*) $\mathcal{R}$ is a set of (conditional) rules $l \to r \Leftarrow C$, where l is a pattern and $Var(r) \subseteq Var(l)$ (*left-linearity*).

A *position* is a sequence $z_1, \ldots, z_n$ of naturals $z_i, i \in I\!N^+$ resp. the empty sequence Λ, identifying t itself; $t_{|p}$ denotes the subterm of t at position p, $t[s]_p$ the result of replacing $t_{|p}$ by s. The concatenation of positions is denoted by $p \cdot q$, $p \preceq q$ means p is a *prefix* of q, i.e. there is a position r with $q = p \cdot r$.

A *reduction step* $t \to_{p,l\to r} t'$ is applicable if there exists a position p, rule $l \to r$ and substitution σ so that $t_{|p} = \sigma(l)$ und $t' = t[\sigma(r)]_p$.

A term t narrows to t' if p is a nonvariable position in t, i.e. $t_{|p} \notin \mathcal{X}$, $l \to r$ variant of a rule in $\mathcal{R}$ with $Var(t) \cap Var(l \to r) = \emptyset$, σ is a unifier of $t_{|p}$ and l, and $t' = \sigma(t[r]_p)$. $t \leadsto_{p,l\to r,\sigma} t'$ is a *narrowing step.*

As rules are applied in one direction only, additional restrictions are necessary to ensure the completeness of narrowing, i.e. the confluence of $\mathcal{R}$. If $l_1 \to r_1$ and $l_2 \to r_2$ are variants and σ is a unifier for l_1 and l_2, then $\sigma(r_1) = \sigma(r_2)$. We deal with infinite data structures, termination of functions is not required. This implies that we have to define equality *on finite objects*, so-called *strict equality* on constructor terms. Let $\wedge$ be a right-associative infix symbol. Strict equality is defined by narrowing to *true* a strict equation $t_1 \approx t_2$.

$$
\begin{array}{ll}
c \approx c \to true & \forall c/0 \in \mathcal{C} \\
c(X_1, \ldots, X_n) \approx c(Y_1, \ldots, Y_n) \to X_1 \approx Y_1 \wedge \ldots \wedge X_n \approx Y_n & \forall c/n \in \mathcal{C}, n > 0 \\
true \wedge X \to X &
\end{array}
$$

A TRS $\mathcal{R}$ is *weakly orthogonal*, if it is left linear and for two rules $l \to r$ and $l' \to r' \in \mathcal{R}$, nonvariable subterm $l_{|p}$ of l and *mgu* σ of $l_{|p}$ and l', σ, also $\sigma(l[r']_p)$ and $\sigma(r)$ are identical.

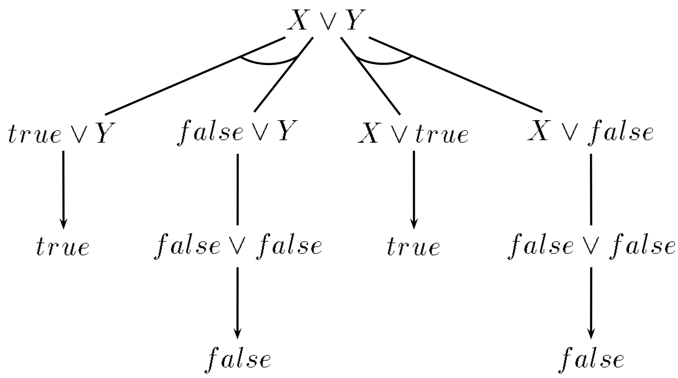

Figure 1: *gdt* for parallel 'or'

The original notion of a (parallel) definitional tree is due to Antoy [1]. Hanus [10] extends this notion as follows. Consider a pattern π and occurrence o of a variable in π. $\mathcal{T}$ is a *partial generalized definitional tree* (*pgdt*) iff one of the following holds:

- $\mathcal{T} = rule(\pi, l \to r)$ with pattern π, l and π unify, i.e. $\pi \to r$ is a variant of a rule in $\mathcal{R}$.
- $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$, where π is a pattern, o is an occurrence of a variable in π, $c_1, \ldots, c_k$ are different constructors of the sort of $\pi_{|o}(k > 0)$ and for $i = 1, \ldots, k, \mathcal{T}_1$ is a generalized definitional tree with pattern $\pi[c_1(x_1, \ldots, x_n)]_o$, where n is the arity of c_i and $x_1, \ldots, x_n$ are new variables.
- $\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$, where $\mathcal{T}_1, \ldots, \mathcal{T}_k$ are generalized definitional trees with pattern π.

$\mathcal{T}$ is a *generalized definitional tree (gdt)* of a n-ary function f, iff $\mathcal{T}$ is a *pgdt* with pattern $f(X_1, \ldots, X_n)$, where n is the arity of f and X_i are fresh variables. For any rule $l \to r$, where $l = f(t_1, \ldots, t_n)$, there exists a leaf $rule(l' \to r')$ in $\mathcal{T}$, where l variant of l'. A *parallel definitional tree* (*prdt*) is a generalized definitional tree in which no subsuming lhs of rules (e.g. $f(X)$ and $f(a)$) are allowed. The strategies described below are defined for *prdts* only.

2.1 Weakly Needed Narrowing

For inductively sequential systems, *needed narrowing* is an optimal strategy [3]. The naive extension to weakly orthogonal systems is known as *weakly needed narrowing* (see [2] for details). As in [10], we use an informal description.

- $\mathcal{T} = rule(\pi \to r)$: apply rule $(\pi \to r)$ to t (t instance of π).
- $\mathcal{T} = branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k)$. For subterm $t_{|o}$ the possibilities are as follows:
  1. $t_{|o}$ bears a defined function symbol at its root: recursively apply strategy λ to $t_{|o}$.
  2. $t_{|o}$ is constructor-rooted: narrow t with $\mathcal{T}_j$, the pattern of which unifies with $t_{|o}$. If no pattern unifies with t, fail.
  3. $t_{|o}$ is a variable: nondeterministically select a subtree $\mathcal{T}_j$, unify t with the pattern of $\mathcal{T}_j$, narrow this instance with $\mathcal{T}_j$.
- $\mathcal{T} = or(\mathcal{T}_1, \ldots, \mathcal{T}_k)$: Nondeterministically select a subtree $\mathcal{T}_j$, proceed narrowing with $\mathcal{T}_j$.

The next example shows that this naive treatment of overlapping rules is inadequate.

Example 2.1 Consider, in addition to the rules of example 1.2, the rule $\neg(\mathit{false}) \to \mathit{true}$ *(R4)*. Weakly needed narrowing yields four derivations for a term $t = \neg(X) \vee \neg(X)$:

$$
\begin{array}{l}
t \rightsquigarrow_{1,R_4,\{X \mapsto false\}} true \vee \neg(false) \rightsquigarrow_{\Lambda,R_2,id} true \\
t \rightsquigarrow_{1,R_4,\{X \mapsto false\}} true \vee \neg(false) \rightsquigarrow_{1,R_4,id} true \vee true \rightsquigarrow_{\Lambda,R_2,id} true \\
t \rightsquigarrow_{2,R_4,\{X \mapsto false\}} \neg(false) \vee true \rightsquigarrow_{\Lambda,R_1,id} true \\
t \rightsquigarrow_{2,R_4,\{X \mapsto false\}} \neg(false) \vee true \rightsquigarrow_{1,R_4,id} true \vee true \rightsquigarrow_{\Lambda,R_2,id} true
\end{array}
$$

It would be more appropriate to narrow several positions simultaneously. □

3 Multistep Narrowing

Example 1.2 has shown that we cannot make a decision which argument to narrow in example 1.2. The classical way to solve the dilemma is trying to exploit determinism (a term $true \vee X$ can be deterministically evaluated to true). Recent approaches were presented in the introduction. They were implemented and examined in some detail from a practical viewpoint in [6]. A completely new approach was taken in [2]. We define a *multistep* informally as the simultaneous reduction of subterms at disjoint positions. *Multistep narrowing* means selecting a suitable substitution, applying it to the term, and executing a parallel rewrite step afterwards.

3.1 The Multistep Implementation

Weakly needed narrowing was implemented both in the straightforward and in the 'parallel' way, where the Prolog rules for a single step reflect the two parts of a narrowing multistep:

```
step(X,Y):- var(X),!,X=Y.
step(f(t1,...,tn),Term2):-
      lambda(f(t1,...,tn),Substs),!,member(Sub,Substs),
      execSubs(Sub,f(t1,...,tn)),
      multistep(f(t1,...,tn),Term1),step(Term1,Term2).         ∀f ∈ Func
step(X,X).
```

A chracteristic of all multistep strategy implementations is the alternation between the collecting of substitutions and the application of a parallel rewrite step[2]. We present the translation for the general case, which we call *multistep narrowing*. Explicit backtracking using the built-in predicate `bagof` becomes necessary, as all *possible* substitutionsare collected:

```
lambda(Term,Sigma):-
      bagof(t(Pos,Subst),collect(Term,NewTerm,Pos,Subst),Tuple),
      makeset(Tuple,Sigma),!.
```

Predicate `makeset` eliminates substitutions that occur more than once. We have to determine a suitable σ first, collecting *all* possible pairs of position and substitution `t(Pos,Subst)`.

```
collect(Term,NewTerm,Pos,Subst):-var(Term),!.
collect(f(t1,...,tn),NewTerm,Pos,Subst):-!,
      f_par(t1,...,tn,NewTerm,Pos,Subst).               ∀f/n ∈ F
collect(c,c,Pos,Subst).                                 ∀c/0 ∈ C
collect(c(X1,...,Xn),c(Y1,...,Yn),Pos,Subst):-
      collect(X1,Y1,Pos1,[1|Pos],Subst).
...
collect(c(X1,...,Xn),c(Y1,...,Yn),Pos,Subst):-
      collect(Xn,Yn,[n|Pos],Subst).                     ∀c/n ∈ C, n > 0
```

[2]For the sake of readability, we abstract from technical details (accumulators) and leave some flags out.

In a multistep, we replace subterms at positions where it is possible and return the unchanged term otherwise.

```
multistep(Term1,Term2):-search(Term1,Term2),!.

search(Term1,Term2):-var(Term1),!,Term1=Term2.
search(c,c).                                              ∀c/0 ∈ C
search(c(X_1,...,X_n),c(Y_1,...,Y_n)):-!,
     search(X_1,Y_1),...,search(X_n,Y_n).                 ∀c/n ∈ C, n > 0
search(f(t_1,...,t_n),Erg):-!,f_ms(t_1,...,t_n,Erg).      ∀f/n ∈ F
```

The following is valid for all multistep strategies. In the leaves of the collection predicates, positions and substitutions are finally known. In the multistep we just return the right hand side.

$trans(rule(f(t_1,\ldots,t_n) \to rhs), p) :=$

```
f_par_p(t_1,...,t_n,Pos,Sub).
```

```
f_ms_p(t_1,...,t_n,rhs).
```

$trans(branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), p) :=$

```
f_par_p(t'_1,...,t'_n,Pos,Sub):-
     collect(Y,Z,Pos,Sub),
     [                              ]
     f_par_{p∪{o}}(t''_1,...,t''_n,Pos,Sub).
```

```
f_ms_p(t'_1,...,t'_n,R):-
     nonvar(Y),f_ms_{p∪{o}}(t_1,...,t_n).
```

```
[f_ms_p(X_1,...,X_n,R):-
     search(X_1,Z_1),...,search(X_n,Z_n),f(Z_1,...,Z_n)=R.]
```

$trans(\mathcal{T}_1, p \cup \{o\})$

...

$trans(\mathcal{T}_k, p \cup \{o\})$

where $\pi = f(t_1,\ldots,t_n)$, $f(t'_1,\ldots,t'_n) := \pi[\mathtt{Y}]_o$, $f(t''_1,\ldots,t''_n) := \pi[\mathtt{Z}]_o$

We explain the piece of code represented by the box[3]. We descend in argument at position o; in case we do not have to descend into another function, and additionally the argument is a variable, we add an explicit binding to the list. With the help of a flag we ensure that, descending in a constructor argument, no new binding is added. Instead of $\{Y \mapsto s(Y_1), Y_1 \mapsto s(Y_2)\}$ we look for $\{Y \mapsto s(s(Y_2))\}$. In appendix A function `h` stands for that case. Otherwise we have an inductively sequential branch, that is the constructor indicated by the definitional tree has to 'match' with the goal argument, $f_{p\cup\{o\}}\ldots$ is called. In the rewrite step, we have to check whether no goal variable has to be bound (built-in `nonvar`). An additional rule, shown in brackets, is created once for every function: if only rules at inner positions are applicable, we descend into *all* arguments, thus obeying the notion of lazyness.

3.2 Parallel Narrowing

Weakly needed narrowing performs redundant computations even in its multistep variant.

Example 3.1 There are two derivations for a parallel 'or' and a goal $\Leftarrow true \vee X \approx true$:

$$\Leftarrow true \vee X \approx true \not\Longrightarrow_{\{id\}} true \quad \text{resp.} \quad \Leftarrow true \vee X \approx true \not\Longrightarrow_{\{X \mapsto true\}} true. \qquad \square$$

Consequently, we have to 'minimize' our set of substitutions as indicated in [18][4].

[3] For technical details see generated code in appendix A.

[4] A formal definition is again found in [2].

- Select a minimal substitution among the non-identity substitutions.
- In addition, substitutions belonging to outer subterms are preferred.

The elimination rules are a verbatim Prolog implementation of the definition given in [2], where also proofs of soundness and completeness can be found. Predicate `lambda` has to be modified slightly, see appendix A. There, code is generated for the following example [2].

Example 3.2 Function f is overlapping, its *gdt* consists of two inductively sequential subtrees.

$f(X, s(Y)) \to s(0)$	$(F1)$	$h(s(s(X))) \to s(0)$	$(H1)$
$f(0, Y) \to s(0)$	$(F2)$	$g(X) \to s(g(X))$	$(G1)$

This is what out implementation returns on a goal `?- f(g(X),f(h(Y),f(0,h(s(Y)))))===Z.`[5]

The collect part yields the list of all posible steps:

```
lambdas:  [t(2.2.2,[_77=s(_626)]),t(2.2,id),t(2.1,[_77=s(s(_575))]),t(1,id)]
```

We minimize this list: $\{Y \mapsto s(s(Y_1))\}$ is subsumed by $\{Y \mapsto s(Y_2)\}$ 2.2 is a prefix of 2.2.2, {`leaving over the identity substitution.`

```
subst. term:  f(g(_47),f(h(_77),f(0,h(s(_77)))))
```

We apply the multistep at positions 1 and $2 \cdot 2$:

```
                         ==> f(s(g(_47)),f(h(_77),s(0)))
```

There are two steps left to narrow the new term to *true*:

```
lambdas:  [t(2,id),t(2.1,[_77=s(s(_1772))]),t(1.1,id)]
minimized:  [id]
subst. term:  f(s(g(_47)),f(h(_77),s(0)))
                         ==> f(s(s(g(_47))),s(0))
lambdas:  [t(Λ,id),t(1.1.1,id)]
minimized:  [id]
subst. term:  f(s(s(g(_47))),s(0))
                              ==> s(0)
Z = s(0) ?  ;
no
```

As σ is always a singleton, the derivation is deterministic, no alternative solutions are found. □

Example 3.3 Parallel narrowing still performs redundant steps on $\Leftarrow (X \vee Y) \vee (true \vee Z) \approx true$.

$$\bar{\lambda} = \{t(1, \mathit{OR1}, X \mapsto true), t(1, \mathit{OR2}, Y \mapsto true), t(2, \mathit{OR1}, id), t(2, \mathit{OR2}, Z \mapsto true)\}$$

Only $Z \mapsto true$ can be eliminated, though just applying the identity would be appropriate. □

3.3 Parallel Narrowing with Simplification

In analogy to *lazy narrowing with simplification* [8], we intend to generate separate pieces of code for a *terminating subset* of the rules. Simplification steps should be applied as long as no goal variable needs to be bound. We use the elimination rules from parallel narrowing. A formal description will appear in the revise version of [2].

3.3.1 Suggested Implementation

We generate predicate `simpstep` in analogy to `step`, to be called *before* every step. Predicate `simpstep` will try to perform as many parallel rewrite steps as possible, while `simpmultistep` corresponds to a `multistep` on the terminating subset.

[5] The debugger outputs positions as lists, the identity substitution is denoted by the empty list.

```
simpstep(f(t_1,...,t_n),R):-simpmultistep(f(t_1,...,t_n),T,F),
     (\+var(F)->simpstep(T,R);T=R).
```

Flag `F` is bound to `step`, if a reduction step is applicable to any term position, otherwise unbound: we return the term and perform a narrowing step. Predicate `step` is modified accordingly.

```
step(f(t_1,...,t_n),R):-simpstep(f(t_1,...,t_n),T),lambda(T,Sub),...
```

Simplification rules are generated with the help of the definitional tree for the terminating rules.

$simptrans(rule(f(t_1,\ldots,t_n) \to rhs), p) :=$

$\boxed{f\texttt{\_ms}_p(t_1\texttt{,}\ldots\texttt{,}t_n\texttt{,rhs).}}$

$simptrans(branch(\pi, o, \mathcal{T}_1, \ldots, \mathcal{T}_k), p) :=$

```
f_ms_p(t'_1,...,t'_n,R):-
     nonvar(Y),f_ms_{p∪{o}}(t_1,...,t_n).
```

```
[f_ms_p(X_1,...,X_n,R):-
     search(X_1,Z_1),...,search(X_n,Z_n),f(Z_1,...,Z_n)=R.]
```

$simptrans(\mathcal{T}_1, p \cup \{o\})$,

...

$simptrans(\mathcal{T}_k, p \cup \{o\})$,

where $\pi = f(t_1,\ldots,t_n), f(t'_1,\ldots,t'_n) := \pi[\texttt{Y}]_o$

All other rules, particularly `simpsearch`, correspond to those of multistep narrowing apart from an additional parameter for the flag.

4 Experimental Results

The search for an optimal strategy is led by two main ideas: exploiting determinism and exploiting expression parallelism. In [6], the influence of programming techniques, purely functional programs, and the elimination of useless computations have been considered in more detail. Let us have a look at two criteria:

- run times for increasing problem sizes[6]
- number of nodes in the narrowing search tree

All strategies based on weakly needed narrowing fall back on needed narrowing for the class of inductively sequential systems, so it is not considered further. Similarly, examining orthognal programs is not too promising as long as we restrict our focus to constructor-based systems[7].

4.1 Comparing Strategies for Weakly Orthogonal Systems

Diagram 2 gives a first orientation about absolute run times. The implementations of the other strategies are out of the scope of this paper [6]. As we mentioned before, implementation overhead of dynamic cut and multistep strategies becomes obvious. The leftmost bars show example 3.2, the second refer to an example where some determinism can be exploited: Berry's system [13].

$berry(0, s(X), Y) \to 0 \qquad berry(X, 0, s(Y)) \to 0 \qquad berry(s(X), Y, 0) \to 0$

[6]Note that measuring run times only is insufficient: Prolog implementations of sophisticated strategies (dynamic cut, multistep strategies) have to elude the evaluation mechanisms of the Prolog compiler. Secondly, run time measurements on multi-user systems are questionable in general. Run times were taken on a 64 MB SPARC-20 under Solaris-2.4.

[7]For interesting theoretical results on general orthogonal systems see [13] [18].

We consider a goal $\Leftarrow berry(A, berry(0, s(B), C), D) \approx E$. Finally, we exploit parallelism extensively in example 1.2 in $\Leftarrow (true \vee (true \vee true)) \vee (X \vee (false \vee false)) \approx true$. Note the implementation overhed of multistep and dynamic cut strategies in a Prolog implementation, whereas run times for BABEL implemented on an abstract machine all remained below $\frac{1}{1000}$ second.

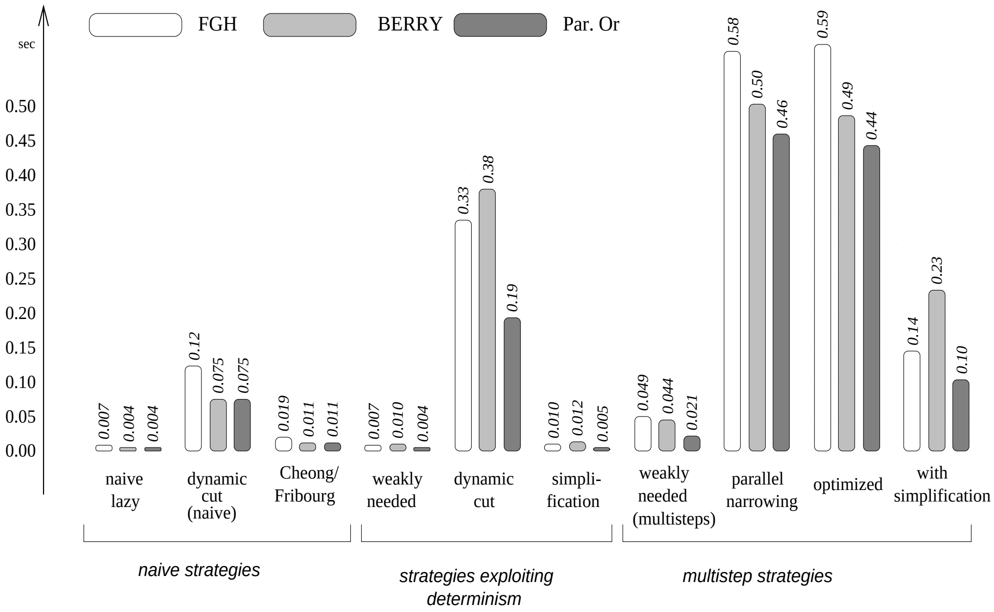

Figure 2: Benchmarks

As pointed out in [8], lazy narrowing with simplification has advantages on generate-and-test programs.

Example 4.1 A typical case is permutation sort on reverted lists. Note that — contrary to all other strategies except lazy narrowing with simplification — parallel narrowing with simplification behaves surprisingly well w.r.t. run times. The implementation overhead is compensated for increasing numbers of elements. The dynamic cut has no effect in this example, both in the Prolog and the BABEL implementation. □

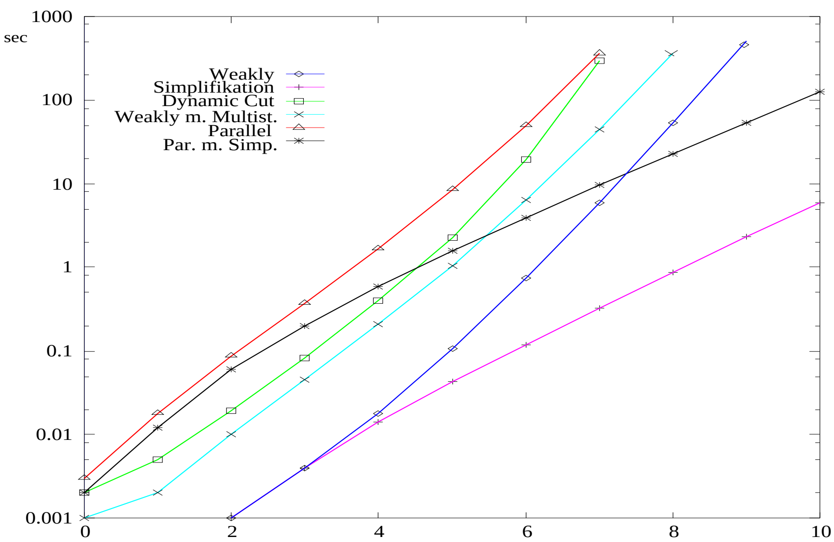

Figure 3: Permutation Sort on Reverted Lists

The exploitation of determinism in other strategies has been discussed in various articles, so we just show the effects on multistep strategies here. It should be obvious that parallel narrowing (with simplification) cuts redundant branches off the search tree for weakly needed narrowing with multisteps the same way lazy narrowing with simplification [8] does. Counting the nodes in the search tree is more appropriate [14].

Example 4.2 Consider again Berry's system together with the following rules:

$$f(0,X,Y) \to 0 \qquad f(X,0,Y) \to 0 \qquad f(X,Y,0) \to 0$$

For $\Leftarrow f(\mathit{berry}(X,Y,Z), \mathit{berry}(0,Y,Z), \mathit{berry}(0,s(Y),Z)) \approx 0$ the number of nodes in the narrowing search tree decreases: substitution $\{Y \mapsto s(Y_1)\}$ subsumes $\{X \mapsto 0, Y \mapsto s(Y_1)\}$ in a parallel narrowing derivation. In parallel narrowing with simplification, as well as in an optimization not discussedhere, the derivation is fully deterministic. $\square$

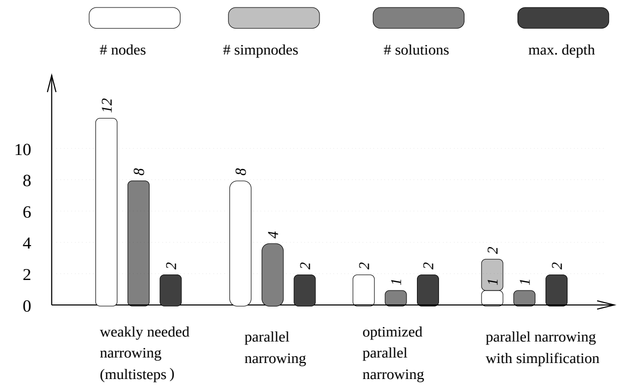

Figure 4: Exploiting Determinism

In a sequential implementation, selected substitutions are considered from left to right. This means, a parallel strategy can only have an advantage over our best sequential strategy (lazy narrowing with simplification), if a tuple from the right eliminates a tuple more to the left and if the substitution in this tuple is not the identity (otherwise, the behaviour would be identical to lazy narrowing with simplification). The following example satisfies both conditions.

Example 4.3 Consider the following rules, describing even numbers and a parallel 'and':

$$\begin{array}{llll} \mathit{even}(0) \to \mathit{true} & (E1) & \mathit{true} \wedge X \to X & X \wedge \mathit{true} \to X \\ \mathit{even}(s(0)) \to \mathit{false} & (E2) & \mathit{false} \wedge X \to \mathit{false} & X \wedge \mathit{false} \to \mathit{false} \\ \mathit{even}(s(s(X)) \to \mathit{even}(X) & (E3) & & \end{array}$$

and a term $\Leftarrow \mathit{even}\ (X) \wedge \mathit{even}\ (s(X))$, returning $\{(1, E1, \{X \mapsto 0\}), (1, E2, \{X \mapsto s(0)\}), (1, E3, \{X \mapsto s(s(Z))\}), (2, E2, \{X \mapsto 0\}), (2, E3, \{X \mapsto s(Z)\})\}$. The elimination of parallel narrowing yields two substitutions: $\bar{\bar{\lambda}} = \{\{X \mapsto 0\}, \{X \mapsto s(Y)\}\}$. Examine the derivation $\Leftarrow \mathit{even}(X) \wedge \mathit{even}(s(X)) \rightsquigarrow_{\{X \mapsto 0\}} \mathit{true} \wedge \mathit{false} \rightsquigarrow_{id} \mathit{false}$. Lazy narrowing with simplification evaluates the arguments sequentially:

$$\Leftarrow \mathit{even}(X) \wedge \mathit{even}(s(X)) \rightsquigarrow_{1,E1,\{X \mapsto 0\}} \mathit{true} \wedge \mathit{even}(s(0))) \to_{2,E1} \mathit{true} \wedge \mathit{false} \to_{1,AND2} \mathit{false}.$$

Obviously, the second derivation is of length 3 (instead of 2). $\square$

However, there are examples where multistep strategies need more steps than hnf-based due to the preference for the identity substitution, so the critrium does not hold in general [6].

Example 4.4 Consider example 1.2 together with a predicate *from* that generates infinite lists. We describe a function *first0* which terminates only if a list starts with 0:

$$\mathit{first0}([s(X)|Xs]) \to \mathit{first0}([s(X)|Xs]) \quad (F1) \qquad \mathit{first0}([0|L]) \to \mathit{true} \quad (F2)$$
$$\mathit{from}(N) \to [N|\mathit{from}(s(N))] \quad (FR)$$

Consider a term $\mathit{first0}(\mathit{from}(s(0))) \vee \mathit{first0}(\mathit{from}(0))$. Lazy narrowing with simplification will cause a nonterminating derivation, because nonterminating rules are only allowed in the narrowing step. However, as we have to evaluate the first argument, no rewrite step can be applied.

$$\mathit{first0}(\mathit{from}(s(0))) \vee \mathit{first0}(\mathit{from}\ (0)) \rightsquigarrow_{1,FR,id} \mathit{first0}([s(0)|\mathit{from}(s(s(0)))]) \vee \mathit{first0}\ (\mathit{from}\ (0))$$
$$\rightsquigarrow_{1,F1,id} \mathit{first0}([s(0)|\mathit{from}(s(s(0)))]) \vee \mathit{first0}(\mathit{from}(0)) \rightsquigarrow_{1,F1,id} \cdots$$

On the other hand, all multistep strategies lead to termination, as the may consider *both* arguments:

$$\mathit{first0}(\mathit{from}(s(0))) \vee \mathit{first0}(\mathit{from}(0)) \Rrightarrow_{id} \mathit{first0}([s(0)|\mathit{from}(s(s(0))]) \vee \mathit{first0}([0|\mathit{from}(s(0))]$$
$$\Rrightarrow_{id} \mathit{first0}([s(0)|\mathit{from}(s(s(0))]) \vee \mathit{true} \to \mathit{true}$$

Lazy narrowing with simplification even has a nondeterministic choice (w.r.t. the rules for $\vee$), contrary to the ideas of functional evaluation. □

We state as a main result of this section that for ground goal, multistep narrowing is optimal in the sense of a Sekar and Ramakrishnan strategy, i.e. among all strategies that consider only the left hand side [2].

5 Other Compiler Features and Outlook

In [6], we suggested a method to extend multistep strategies to conditional equations. This requires the introduction of a *nonambiguity condition* like in BABEL [17]. For convenience, we provide algorithms that extend Antoy's definitional tree creation algorithms [1] to *prdts*. The concrete implementation also comprises the definition of a mini-language CUT (CUrry Translation) and a frontend, so that it should support the development of CURRY [11].

We have presented the first implementation of a narrowing strategy with parallel steps. From the implementation point of view, we have suggested how to implement parallel narrowing with simplification, and have delivered many hints at why this strategy is optimal w.r.t. our criteria. A formal description has yet to be developed, and will appear in a revised version of [2]. We discovered some hints at the effect of *gdt* structure on backtracking behaviour. The formal investigation might require graph theoretical viewpoints and remains for future work. It would also be desirable to extend multistep narrowing to systems described by *generalized* definitional trees, allowing subsuming lhs in rules [10]. Michael Hanus suggested how to simulate such systems using a parallel 'or'. However, while this works well with sequential strategies, it causes overhead with multistep strategies. Problems arise from the necessity to examine the whole term 'breadth-first', so that redundant substitutions may be created in a sequential implementation (see [6]). In that case an if-then-else construct like in BABEL becomes necessary, implying the use of true negation. Run times indicate that the implementation overhead is very high, but likely an implementation on a suitable abstract machine may help.

A Generated Code for Example 3.2

```
lambda(A,B):-bagof(t(C,D),E^F^G^H^collect(A,E,F,[],C,G,[],D,H),I),

multistep(A,B,C):-search(A,B,C),!.

step(A,B):-var(A),!,A=B.
step(f(A,B),C):-lambda(f(A,B),D),!,member(F,D),
      execSubs(F,f(A,B)),multistep(D,G,H),solve(H),step(G,C).
...
step(A,A).
```

```
collect(A,B,C,D,D,E,F,F,G):-var(A),!.
collect(f(A,B),C,D,E,F,G,H,I,J):-!,f_par(A,B,C,D,E,F,G,H,I,J,K).
...
collect(0,0,A,B,B,C,D,D,E).
collect(s(A),s(B),C,D,E,F,G,H,I):-!,collect(A,B,C,[1|D],E,F,G,H,I).

search(A,B,C):-var(A),!,B=A.
search(f(A,B),C,D):-!,f_ms(A,B,C,D).
...
search(0,0,true).
search(s(A),s(B),C):-!,search(A,B,D),C=D.

f_par_or2(A,s(B),s(0),C,D,D,E,F,F,done,G).

f_ms_or2(A,s(B),s(0),true).

f_par(A,B,C,[D|E],F,G,[H|I],J,K,L,M):-collect(B,N,E,[D|F],O,I,J,P,L),
      (nonvar(L)->D=2,K=P,G=O;(var(B)->H=(B=N),Q=[H|J];B=N,Q=J),
      f_par_or2(A,N,C,E,F,G,I,Q,K,L,M)).
f_par(A,B,C,[D|E],F,G,[H|I],J,K,L,M):-collect(A,N,E,[D|F],O,I,J,P,L),
      (nonvar(L)->D=1,K=P,G=O;(var(A)->H=(A=N),Q=[H|J];A=N,Q=J),
      f_par_or1(N,B,C,E,F,G,I,Q,K,L,M)).

f_par_or1(0,A,s(0),B,C,C,D,E,E,done,F).

f_ms_or1(0,A,s(0),true).

f_ms(A,B,C,D):-nonvar(B),f_ms_or2(A,B,C,D).
f_ms(A,B,C,D):-nonvar(A),f_ms_or1(A,B,C,D).
f_ms(A,B,C,D):-(search(A,E,F),search(B,G,H)),D=(F,H),f(E,G)=C.

g_par(A,s(g(A)),B,C,C,D,E,E,done,F).

g_ms(A,s(g(A)),true).
g_ms(A,B,C):-search(A,D,E),C=E,g(D)=B.

h_par(A,B,[C|D],E,F,[G|H],I,J,K,L):-collect(A,M,D,[C|E],N,H,I,O,K),
      (nonvar(K)->C=1,J=O,F=N;(var(A)->G=(A=M),L=no,P=[G|I];A=M,P=O),
      h_par_or1(M,B,D,E,F,H,P,J,K,L)).

h_par_or1(s(A),B,[C|D],E,F,[G|H],I,J,K,L):-collect(A,M,D,[C|E],N,H,I,O,K),
      (nonvar(K)->C=1,J=O,F=N;var(A),L
      h_par_or1s_11(s(M),B,D,E,F,H,P,J,K,L).

h_ms_or1(s(A),B,C):-nonvar(A),h_ms_or1s_11(s(A),B,C).

h_par_or1s_11(s(s(A)),s(0),B,C,C,D,E,E,done,F).

h_ms_or1s_11(s(s(A)),s(0),true).

h_ms(A,B,C):-nonvar(A),h_ms_or1(A,B,C).
h_ms(A,B,C):-search(A,D,E),C=E,h(D)=B.
```

References

[1] S. Antoy, *Definitional Trees*, in: Proc. of the 4th International Conference on Algebraic and Logic Programming, pp. 143-157, Springer LNCS 632 (1992)

[2] S. Antoy, R. Echahed, M. Hanus, *A Parallel Narrowing Strategy*,Technical Report 96/1 (revised version in preparation), Portland State University (1996).

[3] S. Antoy, R. Echahed, M. Hanus, *A Needed Narrowing Strategy*, in: Proc. 21st ACM Symposium on Principles of Programming Languages, pp. 268-279, Portland (1994)

[4] N. Dershowitz, J. Joannaud, *Rewrite Systems*, in: J. van Leeuwen (Hrsg.), Handbook of Theoretical Computer Science B: Formal Methods and Semantics, Kap. 6, pp. 243-320, North Holland, Amsterdam (1990)

[5] L. Fribourg, *SLOG: A Logic Programming Language Interpreter based on Clausal Superposition and Rewriting*, Proc. IEEE Int. Symposium on Logic Programming, pp. 172-184, Boston (1985).

[6] D. Genius, *Comparison and Implementation of Narrowing Strategies in Prolog*, master's thesis (in german), Aachen (1996)

[7] M. Hanus, *Compiling Logic Programs with Equality*, Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming, pp. 387-401, Springer LNCS 456 (1990).

[8] M. Hanus, *Combining Lazy Narrowing and Simplification*, in: Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming, pp. 370-384, Springer LNCS 844, (1994)

[9] M. Hanus, *The Integration of Functions into Logic Programming: from Theory to Practice*, in: Journal of Logic Programming, 19&20, pp. 583-628 (1994)

[10] M. Hanus, *Efficient Compilation of Lazy Narrowing into Prolog*, in: Proc. of the 5th International Workshop on Logic Program Synthesis and Transformation, Utrecht (1995)

[11] M. Hanus, H. Kuchen, J.J. Moreno-Navarro, *Curry: A Truly Functional Logic Language.* Proc. ILPS Workshop on Visions for the Future of Logic Programming, pp. 95-107 (1995)

[12] G. Huet, J. J. Lévy, *Computations in Orthogonal Term Rewriting Systems*, in: J. L. Lassez, G. Plotkin (Hrsg.), *Computational logic: essays in honour of Alan Robinson*, MIT Press, Cambridge, MA (1991); previous version: *Call by Need Computations in Non-ambiguous Linear Term Rewriting Systems*, Technical Report 359, INRIA, Le Chesnay, France (1979)

[13] J. W. Klop, A. Middeldorp, *Sequentiality in Orthogonal Term Rewriting Systems*, in: Journal of Symbolic Computation. pp. 161-195 (1991); previous version: Technical Report CS-R-8932, Stichting Mathematisch Centrum, Amsterdam, Niederlande (1989)

[14] St. Krischer, A. Bockmayr, *Detecting Redundant Narrowing Derivations by the LSE-SL Reducibility Test*, in: Proc. RTA91, LNCS 488, pp. 14-85, Springer, New York (1991)

[15] R. Loogen, St. Winkler, *Dynamic Detection of Determinism in Functional Logic Languages*, in: International Symposium on Programming Language Implementation and Logic Programming (PLILP), pp. 335-346, Springer LNCS 528 (1991)

[16] R. Loogen, F. López Fraguas, M. Rodríguez Artalejo, *A Demand Driven Computation Strategy for Lazy Narrowing*, in: Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming, pp. 184-200. Springer LNCS 714 (1993)

[17] J. J. Moreno-Navarro, M. Rodríguez-Artalejo, *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming, Vol. 12, pp. 191-223 (1992)

[18] R. C. Sekar, I, V. Ramakrishnan, *Programming in Equational Logic: Beyond strong Sequentiality*, in: Information and Computation, 104(1), pp. 78-109, (May 1993)

[19] J. R. Slagle, *Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity*, in: J. ACM 21(4), pp. 622-642 (1974)