Improving Functional Logic Programs by Difference-Lists

Elvira Albert[1], César Ferri[1], Frank Steiner[2], and Germán Vidal[1]

[1] DSIC, UPV, Camino de Vera s/n, E-46022 Valencia, Spain
[2] Institut für Informatik, CAU Kiel, Germany

Abstract. Modern multi-paradigm declarative languages integrate features from functional, logic, and concurrent programming. In this work, we consider the adaptation of the logic programming transformation based on the introduction of difference-lists to an integrated setting. Unfortunately, the use of difference-lists is impractical due to the absence of non-strict equality in lazy (call-by-name) languages. Despite all, we have developed a novel, stepwise transformation which achieves a similar effect over functional logic programs. We also show a simple and practical approach to incorporate the optimization into a real compiler. Finally, we have conducted a number of experiments which show the practicality of our proposal.

Keywords: functional logic programming, program transformation, compiler optimization

1 Introduction

In recent years, several proposals have been made to amalgamate functional and logic programming languages. These languages combine features from functional programming (nested expressions, lazy evaluation) logic programming (logical variables, partial data structures), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). The operational semantics of modern multi-paradigm languages is based on *needed narrowing*, which is currently the best narrowing strategy for lazy functional logic programs due to its optimality properties [4]. Needed narrowing provides completeness in the sense of logic programming (computation of all solutions) as well as functional programming (computation of values), and it can be efficiently implemented by pattern matching and unification.

Example 1. Consider the function $\mathtt{isShorter}$ which is defined by the equations:

$$\begin{array}{lcl}
\mathtt{isShorter}([\,],\mathtt{ys}) & = & \mathtt{True} \\
\mathtt{isShorter}(\mathtt{x}:\mathtt{xs},[\,]) & = & \mathtt{False} \\
\mathtt{isShorter}(\mathtt{x}:\mathtt{xs},\mathtt{y}:\mathtt{ys}) & = & \mathtt{isShorter}(\mathtt{xs},\mathtt{ys})
\end{array}$$

where "[]" and ":" are the constructors of lists. The expression $\mathtt{isShorter}\ (\mathtt{x}:\mathtt{xs})\ \mathtt{z}$ can be evaluated, for instance, by instantiating $\mathtt{z}$ to $(\mathtt{y}{:}\mathtt{ys})$ to apply the third equation, followed by the instantiation of $\mathtt{xs}$ to $[\,]$ to apply the first equation:

$$\mathtt{isShorter}\ (\mathtt{x}:\mathtt{xs})\ \mathtt{z}\ \leadsto_{\{\mathtt{z}\mapsto\mathtt{y}:\mathtt{ys}\}}\ \mathtt{isShorter}\ \mathtt{xs}\ \mathtt{ys}\ \leadsto_{\{\mathtt{xs}\mapsto[\,]\}}\ \mathtt{True}$$

In general, given a term like $\mathtt{isShorter}\ \mathtt{l}_1\ \mathtt{l}_2$, it is always necessary to evaluate $\mathtt{l}_1$ (to some *head normal form*) since all three equations in Example 1 have a non-variable first argument. On the other hand, the evaluation of $\mathtt{l}_2$ is only needed if $\mathtt{l}_1$ is of the form (_:_). Thus, if $\mathtt{l}_1$ is a free variable, needed narrowing instantiates it to a constructor term, here [] or (_:_). Depending on this instantiation, either the first equation is applied or the second argument $\mathtt{l}_2$ is evaluated.

In this work, we consider a well-known list-processing optimization from the logic programming community. Most Prolog programmers know how to use *difference-lists* to improve the efficiency of list-processing programs significatively. Informally, a difference-list is a pair of lists whose second component is a suffix of the first. For example, the list 1:2:[] is encoded as a pair $\langle 1{:}2{:}\mathtt{xs}, \mathtt{xs}\rangle$, where xs is a logical variable. The key to succeed in optimizing programs by difference-lists is the use of a constant-time concatenation: $\mathtt{append}(\langle \mathtt{x},\mathtt{y}\rangle, \langle \mathtt{y},\mathtt{z}\rangle, \langle \mathtt{x},\mathtt{z}\rangle)$. Unfortunately, if we try to adapt this technique to a functional logic context, we find several problems. In particular, a common restriction in lazy functional logic languages is to require *left-linear* rules, i.e., the left-hand sides of the rules cannot contain several occurrences of the same variable. In principle, this restriction does not permit the encoding of concatenation of difference-lists as: $\mathtt{append}*\ \langle \mathtt{x},\mathtt{y}\rangle\ \ \langle \mathtt{y},\mathtt{z}\rangle = \langle \mathtt{x},\mathtt{z}\rangle$ and, consequently, prevents us from having difference-lists in lazy functional languages (at least, at runtime). Therefore, we are interested in a transformation process in which the final program does not contain occurrences of difference-lists. To achieve this goal, we considered that, in some cases, programs using difference-lists are structurally similar to programs written using "accumulating parameters" [12]. Compare, for instance, an optimized version of quicksort by difference-lists (see Sect. 4):

$$\mathtt{qs}*([], \langle \mathtt{ys},\mathtt{ys}\rangle).$$
$$\mathtt{qs}*(\mathtt{x}{:}\mathtt{xs}, \langle \mathtt{ys},\mathtt{ys}'\rangle) :- \mathtt{split}(\mathtt{x},\mathtt{xs},\mathtt{l},\mathtt{r}),\ \mathtt{qs}*(\mathtt{l}, \langle \mathtt{ys},\mathtt{x}{:}\mathtt{w}\rangle),\ \mathtt{qs}*(\mathtt{r}, \langle \mathtt{w},\mathtt{ys}'\rangle).$$

and by introducing accumulating parameters:

$$\mathtt{qs}_{\mathtt{acc}}([], \mathtt{ys}, \mathtt{ys}).$$
$$\mathtt{qs}_{\mathtt{acc}}(\mathtt{x}{:}\mathtt{xs}, \mathtt{ys}', \mathtt{ys}) :- \mathtt{split}(\mathtt{x},\mathtt{xs},\mathtt{l},\mathtt{r}),\ \mathtt{qs}_{\mathtt{acc}}(\mathtt{r},\mathtt{ys}',\mathtt{w}),\ \mathtt{qs}_{\mathtt{acc}}(\mathtt{l},\mathtt{x}{:}\mathtt{w},\mathtt{ys}).$$

We will show that this idea can be generalized, giving rise to an optimization technique which achieves a similar effect over functional logic programs and always returns a program without difference-lists.

The structure of the paper is as follows. After some preliminary definitions in the next section, Sect. 3 describes the language syntax and the operational semantics referenced in our approach. Section 4 introduces a transformation technique (based on the use of difference-lists) which improves a certain class of list-processing programs and shows its correctness and effectiveness. An experimental evaluation of our optimization is shown in Sect. 5. Finally, Sect. 6 presents some related work and Sect. 7 concludes. An extended version of this abstract can be found in [1].

2 Preliminaries

In this section we recall some basic notions from term rewriting [5] and functional logic programming [6]. We consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$

of *constructors* and a set $\mathcal{F}$ of (defined) *functions* or *operations*. There is at least one sort *Bool* containing the constructors `True` and `False`. The set of *constructor terms* with *variables* (e.g., x, y, z) is obtained by using symbols from $\mathcal{C}$ and $\mathcal{X}$. The set of variables occurring in a term t is denoted by $\mathcal{V}ar(t)$. A term t is *ground* if $\mathcal{V}ar(t) = \emptyset$. A term is *linear* if it does not contain multiple occurrences of one variable. We write $\overline{o_n}$ for the *list of objects* $o_1, \ldots, o_n$.

A *pattern* is a term of the form $f(\overline{d_n})$ where $f/n \in \mathcal{F}$ and $d_1, \ldots, d_n$ are constructor terms. A term is *operation-rooted* if it has an operation symbol at the root. A *position* p in a term t is represented by a sequence of natural numbers (Λ denotes the empty sequence, i.e., the root position). $t|_p$ denotes the *subterm* of t at position p, and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term s (see [5] for details).

We denote by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ the *substitution* σ with $\sigma(x_i) = t_i$ for $i = 1, \ldots, n$ (with $x_i \neq x_j$ if $i \neq j$), and $\sigma(x) = x$ for all other variables x. The set $\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\}$ is called the *domain* of σ. A substitution σ is *(ground) constructor*, if $\sigma(x)$ is (ground) constructor for all $x \in \mathcal{D}om(\sigma)$. The identity substitution is denoted by id. Given a substitution θ and a set of variables $V \subseteq \mathcal{X}$, we denote by $\theta_{\restriction V}$ the substitution obtained from θ by restricting its domain to V. We write $\theta = \sigma\ [V]$ if $\theta_{\restriction V} = \sigma_{\restriction V}$, and $\theta \leq \sigma\ [V]$ denotes the existence of a substitution γ such that $\gamma \circ \theta = \sigma\ [V]$.

A set of rewrite rules $l = r$ such that $l \notin \mathcal{X}$, and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ is called a *term rewriting system* (TRS). The terms l and r are called the *left-hand side* and the *right-hand side* of the rule, respectively. A TRS $\mathcal{R}$ is left-linear if l is linear for all $l = r \in \mathcal{R}$. A TRS is constructor-based (CB) if each left-hand side is a pattern. A *rewrite step* is an application of a rewrite rule to a term, i.e., $t \to_{p,R} s$ if there is a position p in t, a rewrite rule $R = (l = r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. In the following, a functional logic *program* is a left-linear CB-TRS.

In order to evaluate terms containing variables, narrowing non-deterministically instantiates the variables so that a rewrite step is possible. Formally, $t \leadsto_{p,R,\sigma} t'$ is a *narrowing step* if p is a non-variable position in t and $\sigma(t) \to_{p,R} t'$. We denote by $t_0 \leadsto^*_\sigma t_n$ a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). Due to the presence of free variables, an expression may be reduced to different values after instantiating free variables to different terms. In functional programming, one is interested in the computed *value* whereas logic programming emphasizes the different bindings (*answers*). In our integrated setting, given a narrowing derivation $t \leadsto^*_\sigma d$ to a constructor term d (possibly with variables), we say that d is the computed value and σ is the computed answer for t.

3 The Language

Modern functional logic languages are based on needed narrowing and inductively sequential programs. Needed narrowing extends the Huet and Lévy's notion of a needed reduction [9]. A precise definition of this class of programs and the needed narrowing strategy is based on the notion of a *definitional tree* [3]. Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves contain all (and only) the rules used to define f and the inner nodes contain information to guide the

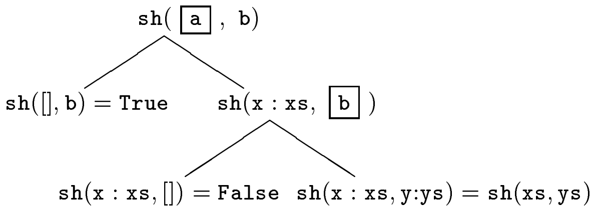

Fig. 1. Definitional tree for `isShorter`

pattern matching during the evaluation of expressions. Each inner node has a *pattern* and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\overline{x_n})$, where $\overline{x_n}$ are different variables. A graphic representation of definitional trees, where each inner node is marked with a pattern, the inductive position in branches is surrounded by a box, and the leaves contain the corresponding rules is often used to illustrate this notion (see, e.g., the definitional tree for the function `isShorter` of Ex. 1 in Fig. 1, here abbreviated as `sh`).

A defined function is called *inductively sequential* if it has a definitional tree. A rewrite system $\mathcal{R}$ is called *inductively sequential* if all its defined functions are inductively sequential. Note that inductively sequential programs are a particular case of left-linear CB-TRSs.

In order to compute needed narrowing steps for an operation-rooted term t, we take a definitional tree $\mathcal{P}$ for the root of t and compute $\lambda(t, \mathcal{P})$. Here, λ is a *narrowing strategy* which returns triples (p, R, σ) containing a position, a program rule, and a substitution. Then, for all $(p, R, \sigma) \in \lambda(t, \mathcal{P})$, $t \leadsto_{p,R,\sigma} t'$ is a *needed narrowing step*. Informally, needed narrowing applies a rule, if possible, otherwise checks the subterm corresponding to the inductive position of the branch: if it is a variable, we instantiate it to the constructor of a child; if it is already a constructor, we proceed with the corresponding child; finally, if it is a function, we evaluate it by recursively applying needed narrowing. For inductively sequential programs, needed narrowing is sound and complete w.r.t. *strict* equations (i.e., both sides must reduce to the same ground constructor term) and constructor substitutions as solutions [4].

4 Optimization by Accumulating Parameters

In this section, we introduce a new transformation for optimizing *functions that independently build different sections of a list to be later combined together* [12]. The development of this section is inspired by the well-known difference-list transformation from the logic programming community [11, 12].

The idea behind the difference-list transformation of [11] is to replace certain lists by terms called *difference-lists* in order to expose opportunities for a faster concatenation. A difference-list is represented as a pair of lists whose second component is a suffix of the first. For example, the list 1:2:[] is encoded as a pair $\langle$1:2:`xs`, `xs`$\rangle$, where `xs` is a logical variable. Therefore, a difference-list represents the list which results from removing the suffix from the first component. Informally, a difference-list can be seen as a "list plus a pointer to its tail". By virtue of the new representation, such a pointer may avoid traversing some lists represented by difference-lists, since the concatenation of difference-lists is a constant-time operation: $\texttt{append\_dl}(\langle x,y\rangle, \langle y,z\rangle, \langle x,z\rangle)$. Therefore, predicates using append_dl take advantage from its improved runtime, as we now illustrate by considering the quicksort algorithm:

$$\begin{array}{l} \texttt{qs}([],[]). \\ \texttt{qs}(x{:}xs, ys)\ :-\ \texttt{split}(x,xs,l,r),\ \texttt{qs}(l,z),\ \texttt{qs}(r,w),\ \texttt{append}(z, x{:}w, ys). \end{array}$$

The definition of the predicate split is not relevant here, it is sufficient to know that, given a call $\texttt{split}(x,xs,l,r)$ it returns in l all the elements of the list xs which are lesser than x and in r those which are greater than x. Following [11], the second argument of qs and all the arguments of append need to be changed to difference-lists by using the equivalences:

$$\begin{array}{ll} [] & \rightarrow \langle y,y\rangle \\ t_1{:}\ldots{:}t_n{:}[] & \rightarrow \langle t_1{:}\ldots{:}t_n{:}y, y\rangle \\ x & \rightarrow \langle x,y\rangle \end{array}$$

where y is a fresh variable. Thus, we obtain the program:

$$\begin{array}{l} \texttt{qs}(xs,ys)\ :-\ \texttt{qs*}(xs,\langle ys,[]\rangle). \\ \texttt{qs*}([],\langle ys,ys\rangle). \\ \texttt{qs*}(x{:}xs,\langle ys,ys'\rangle)\ :-\ \texttt{split}(x,xs,l,r),\ \texttt{qs*}(l,\langle z,zs\rangle),\ \texttt{qs*}(r,\langle w,ws\rangle), \\ \qquad\qquad\qquad\qquad\qquad \texttt{append\_dl}(\langle z,zs\rangle, \langle x{:}w,ws\rangle, \langle ys,ys'\rangle). \end{array}$$

Note that the first rule is introduced to relate the new predicate qs∗ and the original qs (since the difference-list $\langle ys,[]\rangle$ is equivalent to the standard list ys). By unfolding the call to append_dl, we get an improved definition of qs:

$$\begin{array}{l} \texttt{qs}(xs,ys)\ :-\ \texttt{qs*}(xs,\langle ys,[]\rangle). \\ \texttt{qs*}([],\langle ys,ys\rangle). \\ \texttt{qs*}(x{:}xs,\langle ys,ys'\rangle) :-\ \texttt{split}(x,xs,l,r),\ \texttt{qs*}(l,\langle ys,x{:}w\rangle),\ \texttt{qs*}(r,\langle w,ys'\rangle). \end{array}$$

In an attempt to adapt this technique to a functional logic context, we find several problems. In particular, a common restriction in lazy functional logic languages is to require *left-linear* rules, i.e., the left-hand sides of the rules cannot contain several occurrences of the same variable. In principle, this restriction prevents us from encoding the concatenation of difference-lists as a rule of the form: $\texttt{append*}(\langle x,y\rangle, \langle y,z\rangle) = \langle x,z\rangle$. Of course, we can transform it into:

$$\texttt{append\_dl}(\langle x,y\rangle, \langle w,z\rangle)\ |\ \ y == w\ \ =\ \ \langle x,z\rangle$$

by using a guarded expression. However, in order to keep the effectiveness of the transformation, the equality symbol "==" should be interpreted as syntactic unification, which is not allowed in lazy functional logic programs where only strict equality is permitted. In general, the manipulation of difference-lists requires the use of non-strict equality in order to *assign* terms to the pointers of difference-lists. Therefore, we are interested in a transformation process in which the final program does not contain occurrences of difference-lists (nor calls to append∗).

To achieve this goal, we considered that, in some cases, programs using difference-lists are structurally similar to programs written using accumulators. For instance, quicksort can be defined using accumulators as follows:

$$\begin{array}{ll}
\mathtt{qs}(\mathtt{xs},\mathtt{ys}) & :-\ \mathtt{qs}_{\mathtt{acc}}(\mathtt{xs},[],\mathtt{ys}). \\
\mathtt{qs}_{\mathtt{acc}}([],\mathtt{ys},\mathtt{ys}). & \\
\mathtt{qs}_{\mathtt{acc}}(\mathtt{x}{:}\mathtt{xs},\mathtt{ys}',\mathtt{ys}) & :-\ \mathtt{split}(\mathtt{x},\mathtt{xs},\mathtt{l},\mathtt{r}),\ \mathtt{qs}_{\mathtt{acc}}(\mathtt{r},\mathtt{ys}',\mathtt{w}),\ \mathtt{qs}_{\mathtt{acc}}(\mathtt{l},\mathtt{x}{:}\mathtt{w},\mathtt{ys}).
\end{array}$$

There are only two differences between this program and the difference-list version. The first difference is syntactic: the difference-list is represented as two independent arguments, but in reverse order, the tail preceding the head. The second difference is the goal order in the body of the recursive clause of $\mathtt{qs}_{\mathtt{acc}}$. The net effect is that the sorted list is built *bottom-up* from its tail, rather than *top-down* from its head [12].

Now we show, by means of an example, an adaptation of the difference-list transformation to a functional logic language.

4.1 An Example of the Difference-lists Transformation

Consider again the quicksort algorithm, but now with a functional (logic) syntax:

$$\begin{array}{ll}
\mathtt{qs}([]) & = [] \\
\mathtt{qs}(\mathtt{x}{:}\mathtt{xs}) & = \mathtt{append}(\mathtt{qs}(\mathtt{l}),\mathtt{x}{:}\mathtt{qs}(\mathtt{r}))\ \ \mathtt{where}\ (\mathtt{l},\mathtt{r}) = \mathtt{split}(\mathtt{x},\mathtt{xs})
\end{array}$$

Here, both `qs` and `split` are the functional counterpart of the predicates used in the previous section.

As dictated by the method of [11], the three arguments of the predicate `append` as well as the second argument of the predicate `qs` should be changed by difference-lists. Similarly, in our functional syntax, we will replace the arguments of the function `append` and the *result* of both functions by difference-lists. From the previous section, we know how to transform different kinds of standard lists into difference-lists; now, however, we are faced with a new situation which arises the question: how can we transform an operation-rooted term into a difference-list? To solve this problem, we allow the *flattening* of some calls by using a sort of conditional expressions. The main difference with standard guarded expressions is that, in order to preserve the semantics, we use a syntactic (non-strict) equality "$\approx$" for the equations in the condition. In this way, we get the following transformed program:

$$\begin{array}{ll}
\mathtt{qs}(\mathtt{x}) & = \mathtt{y}\ \Leftarrow\ \langle\mathtt{y},[]\rangle \approx \mathtt{qs{*}}(\mathtt{x}) \\
\mathtt{qs{*}}([]) & = \langle\mathtt{x},\mathtt{x}\rangle \\
\mathtt{qs{*}}(\mathtt{x}{:}\mathtt{xs}) & = \mathtt{append{*}}(\langle\mathtt{z},\mathtt{zs}\rangle,\langle\mathtt{x}{:}\mathtt{w},\mathtt{ws}\rangle)\ \Leftarrow\ \langle\mathtt{z},\mathtt{zs}\rangle \approx \mathtt{qs{*}}(\mathtt{l}), \langle\mathtt{w},\mathtt{ws}\rangle \approx \mathtt{qs{*}}(\mathtt{r}) \\
 & \quad\ \mathtt{where}\ (\mathtt{l},\mathtt{r}) = \mathtt{split}(\mathtt{x},\mathtt{xs})
\end{array}$$

By defining the constant-time `append*` by the rule: $\mathtt{append{*}}(\langle\mathtt{x},\mathtt{y}\rangle,\langle\mathtt{y},\mathtt{z}\rangle) = \langle\mathtt{x},\mathtt{z}\rangle$, we can unfold the calls to `append*` as follows:

$$\begin{array}{ll}
\mathtt{qs}(\mathtt{x}) & = \mathtt{y}\ \Leftarrow\ \langle\mathtt{y},[]\rangle \approx \mathtt{qs{*}}(\mathtt{x}) \\
\mathtt{qs{*}}([]) & = \langle\mathtt{x},\mathtt{x}\rangle \\
\mathtt{qs{*}}(\mathtt{x}{:}\mathtt{xs}) & = \langle\mathtt{z},\mathtt{ws}\rangle\ \Leftarrow\ \langle\mathtt{z},\mathtt{x}{:}\mathtt{w}\rangle \approx \mathtt{qs{*}}(\mathtt{l}), \langle\mathtt{w},\mathtt{ws}\rangle \approx \mathtt{qs{*}}(\mathtt{r}) \\
 & \quad\ \mathtt{where}\ (\mathtt{l},\mathtt{r}) = \mathtt{split}(\mathtt{x},\mathtt{xs})
\end{array}$$

In contrast to [11], now we want to remove difference-lists from the program. Intuitively, the idea is to detect that, since we only allow difference-lists in the result of functions, the second argument of the difference-list is somehow used to construct the final result progressively and, thus, we can change it by an "accumulating parameter".

Also, since the calls to $\mathtt{qs}*$ are flattened using a conditional expression, we need to move the second argument of the difference-list to the corresponding call to $\mathtt{qs}*$:

$$
\begin{array}{lll}
\mathtt{qs(x)} & = \mathtt{y} \ \Leftarrow\ \mathtt{y} \approx \mathtt{qs_{acc}(x,[])} \\
\mathtt{qs_{acc}([],x)} & = \mathtt{x} \\
\mathtt{qs_{acc}(x{:}xs,ws)} & = \mathtt{z} \ \Leftarrow\ \mathtt{z} \approx \mathtt{qs_{acc}(l,x{:}w)},\ \mathtt{w} \approx \mathtt{qs_{acc}(r,ws)} \\
 & \quad \mathtt{where\ (l,r) = split(x,xs)}
\end{array}
$$

where $\mathtt{qs}*$ is renamed as $\mathtt{qs_{acc}}$. Finally, by simplifying the equations in the conditions (i.e., by unifying them), we achieve the desired optimization:

$$
\begin{array}{ll}
\mathtt{qs(x)} & = \mathtt{qs_{acc}(x,[])} \\
\mathtt{qs_{acc}([],x)} & = \mathtt{x} \\
\mathtt{qs_{acc}(x{:}xs,ws)} & = \mathtt{qs_{acc}(l,x{:}qs_{acc}(r,ws))\ where\ (l,r) = split(x,xs)}
\end{array}
$$

which gives a similar improvement as the optimized predicate $\mathtt{qs}*$ above. Indeed, thanks to the use of accumulating parameters, we avoid the traversal of the list computed by $\mathtt{qs(l)}$ on each recursive call. In general, this optimization is able to produce superlinear speedups [11, 12].

4.2 The Stepwise Transformation

(a) Marking Algorithm:
Given a function to be optimized, a marking algorithm is applied in order to determine which expressions need to be replaced by difference-lists.

1. Input: a program $\mathcal{R}$ and a function $\mathtt{f}$ whose result type is a list
2. Initialization: $\mathcal{M}_0 = \{\mathtt{f}\}$, $i = 0$
3. Repeat
   - for each function in $\mathcal{M}_i$, mark the right-hand sides of the rules defining $\mathtt{f}$
   - propagate marks among expressions by applying the following rules:

     $$
     \begin{array}{ll}
     \underline{\mathtt{append}(t_1,t_2)} & \to \mathtt{append}(\underline{t_1},\underline{t_2}) \\
     \underline{t_1 : t_2} & \to t_1 \ \underline{:}\ \underline{t_2} \\
     \underline{\mathtt{g}(t_1,\ldots,t_n)} & \to \underline{\mathtt{g}}(t_1,\ldots,t_n)
     \end{array}
     $$

     where $\mathtt{g} \in \mathcal{F}$ is a defined function symbol different from $\mathtt{append}$.
   - if there is a marked expression $\underline{t}$ such that t is a variable, then return FAIL; else $\mathcal{M}_{i+1} = \{\mathtt{h} \mid \underline{\mathtt{h}}(t_1,\ldots,t_k) \text{ appears in } \mathcal{R}\}$

   until $\mathcal{M}_i = \mathcal{M}_{i+1}$

(b) Introduction of Difference-lists:
If the marking algorithm does not return FAIL, then we use the function τ to transform expressions rooted by a marked symbol into difference-lists:

$$
\begin{array}{ll}
\tau([]) & = \langle y,y\rangle \\
\tau(t_1{:}t_2) & = \langle t_1{:}s,s'\rangle \quad \text{where } \langle s,s'\rangle := \tau(t_2) \\
\tau(\mathtt{f}(\overline{t_n})) & = \langle y,y'\rangle \ \Leftarrow\ \langle y,y'\rangle \approx \mathtt{f}(\overline{t_n})
\end{array}
$$

where y, y' are fresh variables not appearing in the program and those occurrences of `append` whose arguments have been replaced by difference-lists are renamed as `append*`. Furthermore, we consider that all marked function symbols `f` in the resulting program are replaced by `f*`. For instance, a rule of the form $\mathtt{f}(\overline{\mathtt{c_n}}) = \mathtt{t_1} \,\underline{:}\, \mathtt{t_2} \,\underline{:}\, \underline{\mathtt{f}}(\overline{\mathtt{s_n}})$ is transformed into:

$$\mathtt{f*}(\overline{\mathtt{c_n}}) = \langle \mathtt{t_1} : \mathtt{t_2} : \mathtt{y}, \mathtt{y'} \rangle \;\Leftarrow\; \langle \mathtt{y}, \mathtt{y'} \rangle \approx \mathtt{f*}(\overline{\mathtt{s_n}})$$

As illustrated in the example of Sect. 4.1, when the transformation of several terms gives rise to conditional expressions, all the equations are joined into a single condition. The following equation replaces the original definition of `f`:

$$\mathtt{f}(\overline{x_n}) \;=\; y \;\Leftarrow \langle y, [] \rangle \approx \mathtt{f*}(\overline{x_n})$$

Let us remark that the introduction of non-strict equalities does not destroy the correctness of the transformation, since they can be seen as a technical artifice in this stage but will be removed from the program in stage (e).

(c) Unfolding of `append*`:

The next step consists in unfolding[1] the calls to `append*` using the following rule:

$$\mathtt{append*}(\langle \mathtt{x}, \mathtt{y} \rangle \,, \langle \mathtt{y}, \mathtt{z} \rangle) = \langle \mathtt{x}, \mathtt{z} \rangle \;\; .$$

Note that this rule is not legal in a functional logic language. It is used during the transformation but no calls to `append*` appear in the final program.

(d) Use of Accumulating Parameters:

Then, we move the second argument of difference-lists to the corresponding function call as indicated by these rules:

$$[\mathtt{f*}(\overline{t_n}) = \langle y, y' \rangle \;\Leftarrow\; C] \to [\mathtt{f}_{\mathtt{acc}}(\overline{t_n}, y') = y \;\Leftarrow\; C]$$
$$[t \;\Leftarrow\; \langle s, s' \rangle \approx \mathtt{f*}(\overline{t_n})] \;\;\to [t \;\Leftarrow\; s \approx \mathtt{f}_{\mathtt{acc}}(\overline{t_n}, s')]$$

This corresponds to the idea of converting the second argument of the difference-lists in an accumulating parameter of the function in which the result will be computed.

(e) Simplification:

The final step of the transformation simplifies further the program by unfolding the (non-strict) equations in the conditional expressions (i.e., by unifying them). In this way, we guarantee that all conditional expressions are removed from the program, since the first argument of difference-lists is always a free variable.

Let us illustrate how our strategy proceeds with two examples. As an example of complete transformation, consider the following contrived example, which we use to illustrate the actions taken by each stage:

$$\begin{aligned} \mathtt{f}([], \mathtt{y}) &= \mathtt{y}{:}[] \\ \mathtt{f}(\mathtt{x}{:}\mathtt{xs}, \mathtt{y}) &= \mathtt{append}(\mathtt{f}(\mathtt{xs}, \mathtt{y}), \mathtt{x}{:}\mathtt{g}(\mathtt{xs})) \\ \mathtt{g}([]) &= [] \\ \mathtt{g}(\mathtt{x}{:}\mathtt{xs}) &= \mathtt{x}{:}\mathtt{g}(\mathtt{xs}) \end{aligned}$$

[1] In particular, we use an unfolding similar to [13], but using (needed) narrowing instead of SLD-resolution (as defined in [2]).

If we start the marking algorithm with function f, we get the marked program:

$$
\begin{aligned}
\mathtt{f}([],\mathtt{y}) &= \mathtt{y} \,\underline{:}\, \underline{[]}\\
\mathtt{f}(\mathtt{x{:}xs},\mathtt{y}) &= \mathtt{append}(\underline{\mathtt{f}}(\mathtt{xs},\mathtt{y}),\mathtt{x} \,\underline{:}\, \underline{\mathtt{g}}(\mathtt{xs}))\\
\mathtt{g}([]) &= \underline{[]}\\
\mathtt{g}(\mathtt{x{:}xs}) &= \mathtt{x} \,\underline{:}\, \underline{\mathtt{g}}(\mathtt{xs})
\end{aligned}
$$

After replacing the marked expressions by difference-lists:

$$
\begin{aligned}
\mathtt{f}(\mathtt{x},\mathtt{y}) &= \mathtt{z} \ \Leftarrow\ \langle \mathtt{z},[]\rangle \approx \mathtt{f*}(\mathtt{x},\mathtt{y})\\
\mathtt{f*}([],\mathtt{y}) &= \langle \mathtt{y{:}z},\mathtt{z}\rangle\\
\mathtt{f*}(\mathtt{x{:}xs},\mathtt{y}) &= \mathtt{append*}(\langle \mathtt{z},\mathtt{z}'\rangle\,,\langle \mathtt{x{:}w},\mathtt{w}'\rangle) \ \Leftarrow\ \langle \mathtt{z},\mathtt{z}'\rangle \approx \mathtt{f*}(\mathtt{xs},\mathtt{y}),\langle \mathtt{w},\mathtt{w}'\rangle \approx \mathtt{g*}(\mathtt{xs})\\
\mathtt{g}(\mathtt{x}) &= \mathtt{y} \ \Leftarrow\ \langle \mathtt{y},[]\rangle \approx \mathtt{g*}(\mathtt{x})\\
\mathtt{g*}([]) &= \langle \mathtt{y},\mathtt{y}\rangle\\
\mathtt{g*}(\mathtt{x{:}xs}) &= \langle \mathtt{x{:}y},\mathtt{y}'\rangle \ \Leftarrow\ \langle \mathtt{y},\mathtt{y}'\rangle \approx \mathtt{g*}(\mathtt{xs})
\end{aligned}
$$

By unfolding the call to append∗:

$$
\begin{aligned}
&\vdots\\
\mathtt{f*}(\mathtt{x{:}xs},\mathtt{y}) &= \langle \mathtt{z},\mathtt{w}'\rangle \ \Leftarrow\ \langle \mathtt{z},\mathtt{x{:}w}\rangle \approx \mathtt{f*}(\mathtt{xs},\mathtt{y}),\ \langle \mathtt{w},\mathtt{w}'\rangle \approx \mathtt{g*}(\mathtt{xs})\\
&\vdots
\end{aligned}
$$

By introducing accumulating parameters:

$$
\begin{aligned}
\mathtt{f}(\mathtt{x},\mathtt{y}) &= \mathtt{z} \ \Leftarrow \mathtt{z} \approx \mathtt{f}_{\mathtt{acc}}(\mathtt{x},\mathtt{y},[])\\
\mathtt{f}_{\mathtt{acc}}([],\mathtt{y},\mathtt{z}) &= \mathtt{y{:}z}\\
\mathtt{f}_{\mathtt{acc}}(\mathtt{x{:}xs},\mathtt{y},\mathtt{w}') &= \mathtt{z} \ \Leftarrow \mathtt{z} \approx \mathtt{f}_{\mathtt{acc}}(\mathtt{xs},\mathtt{y},\mathtt{x{:}w}),\mathtt{w} \approx \mathtt{g}_{\mathtt{acc}}(\mathtt{xs},\mathtt{w}')\\
\mathtt{g}(\mathtt{x}) &= \mathtt{y} \ \Leftarrow \mathtt{y} \approx \mathtt{g}_{\mathtt{acc}}(\mathtt{x},[])\\
\mathtt{g}_{\mathtt{acc}}([],\mathtt{y}) &= \mathtt{y}\\
\mathtt{g}_{\mathtt{acc}}(\mathtt{x{:}xs},\mathtt{y}') &= \mathtt{x{:}y} \ \Leftarrow \mathtt{y} \approx \mathtt{g}_{\mathtt{acc}}(\mathtt{xs},\mathtt{y}')
\end{aligned}
$$

Finally, by unfolding the conditions, we get:

$$
\begin{aligned}
\mathtt{f}(\mathtt{x},\mathtt{y}) &= \mathtt{f}_{\mathtt{acc}}(\mathtt{x},\mathtt{y},[])\\
\mathtt{f}_{\mathtt{acc}}([],\mathtt{y},\mathtt{z}) &= \mathtt{y{:}z}\\
\mathtt{f}_{\mathtt{acc}}(\mathtt{x{:}xs},\mathtt{y},\mathtt{w}') &= \mathtt{f}_{\mathtt{acc}}(\mathtt{xs},\mathtt{y},\mathtt{x}{:}\mathtt{g}_{\mathtt{acc}}(\mathtt{xs},\mathtt{w}'))\\
\mathtt{g}(\mathtt{x}) &= \mathtt{g}_{\mathtt{acc}}(\mathtt{x},[])\\
\mathtt{g}_{\mathtt{acc}}([],\mathtt{y}) &= \mathtt{y}\\
\mathtt{g}_{\mathtt{acc}}(\mathtt{x{:}xs},\mathtt{y}') &= \mathtt{x}{:}\mathtt{g}_{\mathtt{acc}}(\mathtt{xs},\mathtt{y}')
\end{aligned}
$$

Intuitively, the effect of the transformation is that, in the resulting program, the operations over the input list to f are mixed up, while in the original program they were built independently (and then combined by the function append).

As an example of program to which the transformation cannot be applied, consider the double append program:

$$\mathtt{dapp}(\mathtt{x},\mathtt{y},\mathtt{z}) = \mathtt{append}(\mathtt{append}(\mathtt{x},\mathtt{y}),\mathtt{z})$$

If we start the marking algorithm with the function dapp, in the first iteration we get the following marked program:

$$\mathtt{dapp}(\mathtt{x},\mathtt{y},\mathtt{z}) = \mathtt{append}(\mathtt{append}(\underline{\mathtt{x}},\underline{\mathtt{y}}),\underline{\mathtt{z}})$$

Therefore, stage (a) incurs into FAIL since the variables x, y, and z have been marked.

Note that by allowing stage (b) (as it actually happens in the original difference-list transformation), we would obtain the following definition of `dapp`:

$$\texttt{dapp}(\langle \texttt{x}, \texttt{xs}\rangle, \langle \texttt{xs}, \texttt{ys}\rangle, \langle \texttt{ys}, \texttt{z}\rangle) \; = \; \langle \texttt{x}, \texttt{z}\rangle \; .$$

However, stage (c) could not remove the difference-lists of the arguments of `dapp` and, thus, we would produce a non-legal program.

Notice that, even if the marking algorithm does not return FAIL, improvement is not guaranteed (although there is no significant loss of efficiency in these cases, see the function `g` in the example above). In order to always guarantee runtime improvement, stage (a) is only started with functions whose definitions are of the form $\texttt{append}(t_1, t_2)$; this way we ensure that, if the method is actually applied, at least one call to `append` from each of them will be removed and, consequently, some gain will be achieved.

4.3 Correctness

The correctness of the transformation can be derived from the correctness of stages (b) and (d), since the remaining stages do not modify the program —stage (a)— or are instances of the fold/unfold framework of [2] —stages (c) and (e). In the following, we develop a proof sketch for stages (b) and (d) under certain conditions on the form of difference-lists (i.e., only *lazily* regular lists are allowed in the first argument of `append*`, see below).

To prove the correctness of stage (b), we first need to define an adequate semantics for conditional expressions in transformed programs. Basically, it can be provided as follows. Let us consider an initial (marked) program $\mathcal{R}_a$ and the program $\mathcal{R}_b$ obtained from applying stage (b) to $\mathcal{R}_a$. Now, we introduce the following function τ':

$$\begin{array}{ll} \tau'([]) & = [] \\ \tau'(t_1{:}t_2) & = t_1{:}\tau'(t_2) \\ \tau'(f(\overline{t_n})) & = y \;\Leftarrow\; y \approx f(\overline{t_n}) \end{array}$$

which is used to transform the initial program $\mathcal{R}_a$ into a modified version $\mathcal{R}'_a$ with the same structure than $\mathcal{R}_b$ (but without difference-lists). It should be clear that each needed narrowing derivation in $\mathcal{R}_a$ can be mimicked in $\mathcal{R}'_a$, since the only difference is that some expressions containing nested function symbols have been flattened into (non-strict) equalities. This way, we can define the semantics of conditional expressions in $\mathcal{R}_b$ in terms of the associated needed narrowing steps in the original program $\mathcal{R}_a$ (via the equivalence with $\mathcal{R}'_a$). Furthermore, when evaluating terms in $\mathcal{R}_b$, we allow the flattening of expressions, as well as the unfolding of equations, in order to preserve the equivalence with the computations in $\mathcal{R}_a$.

Once the interpretation of conditional expressions is fixed, we can establish the following equivalence between derivations in $\mathcal{R}_a$ and $\mathcal{R}_b$ where no call to `append` occurs. Given an operation-rooted term $e = \texttt{f}(t_1, \ldots, t_n)$ such that `f` is marked by the algorithm in stage (a), then

$$e \leadsto^*_\sigma d{:}[] \;\text{ in } \mathcal{R}_a \;\text{ iff }\; e' \leadsto^*_{\sigma'} \langle d{:}y, y\rangle \;\text{ in } \mathcal{R}_b \qquad (*)$$

where $e' = \texttt{f*}(t_1, \ldots, t_n)$, $\sigma = \sigma'\ [\mathcal{V}ar(e)]$, and d represents a (possibly empty) sequence of elements of the form $d_1{:}\ldots{:}d_k$, $k \geq 0$. Note that, by the definition of the marking algorithm, the terms $t_1, \ldots, t_n$ cannot contain marked function symbols. This equivalence can be easily stated by induction on the length of the derivations, by considering these three facts: i) no calls to `append` (resp. `append*`) are produced in the first (resp. the second) derivation; ii) the left-hand sides of the applied rules are the same in both derivations since they are not changed by stage (b); and iii) the modifications in the right-hand sides can be easily proven from the equivalence between lists and difference-lists and the interpretation of conditional expressions. Therefore, we center the discussion on the correctness of the function `append*`.

In [11], the notion of *regular* difference-list is introduced to ensure the correctness of `append*`; namely, only calls to `append*` with a regular difference-list in the first argument are allowed. Essentially, a difference-list is regular if it is of the form $\langle t_1 \!:\! \ldots \!:\! t_n \!:\! y, y\rangle$ and y does not appear in $t_1, \ldots, t_n$, i.e., if it denotes a finite list (here $t_1 \!:\! \ldots \!:\! t_n \!:\! [\,]$). This notion of regularity is not appropriate in our context due to lazy evaluation, since we can have calls to `append*` with a non-regular difference-list in the first argument, and still preserve correctness if this argument is evaluated to a regular difference-list afterwards. To overcome this restriction (which drastically would reduce the number of programs amenable to be transformed), we introduce a lazy version of regular list as follows. Given an expression $e[d_1]_p$ containing a difference-list d_1 at some position p, we say that d_1 is *lazily* regular in a derivation $e[d_1]_p \leadsto^*_\sigma e'$ iff $\sigma(d_1)$ is regular (i.e., of the form $\langle t_1 \!:\! \ldots \!:\! t_n \!:\! y, y\rangle$). Now, by using the notion of lazily regular lists, we can state the correctness of `append*` as follows.[2] Let e_1, e_2 be expressions with no calls to `append` and let e'_1, e'_2 be the corresponding expressions which result from replacing each call to a marked function `f` by the corresponding call `f*`. Then,

$$\begin{array}{c}\texttt{append}(e_1, e_2) \leadsto^*_\sigma d \!:\! \sigma(e_2) \ \text{ in } \mathcal{R}_a\\ \text{iff}\end{array}$$

$$\begin{array}{l}\texttt{append*}(\langle x, xs\rangle, \langle y, ys\rangle) \ \Leftarrow \ \langle x, xs\rangle \approx e'_1, \langle y, ys\rangle \approx e'_2\\ \qquad \leadsto^*_{\sigma'} \langle d \!:\! y, ys\rangle \ \Leftarrow \ \langle y, ys\rangle \approx \sigma'(e'_2) \ \text{ in } \mathcal{R}_b\end{array}$$

where e_1 is lazily regular in the second derivation, $\sigma = \sigma' \ [\mathcal{V}ar(\{e_1, e_2\})]$, and d represents a (possibly empty) sequence of elements of the form $d_1 \!:\! \ldots \!:\! d_k$, $k \geq 0$.

Let us prove the claim by considering both implications:
($\Rightarrow$) Consider the derivation $\texttt{append}(e_1, e_2) \leadsto^*_\sigma d \!:\! \sigma(e_2)$ in $\mathcal{R}_a$. By definition of needed narrowing, it is immediate that $e_1 \leadsto^*_\sigma d \!:\! [\,]$. By equivalence $(*)$, we have $e'_1 \leadsto^*_{\sigma'} \langle d \!:\! z, z\rangle$ in $\mathcal{R}_b$, where $\sigma = \sigma' \ [\mathcal{V}ar(e_1)]$. Therefore,

$$\begin{array}{lrcl}\texttt{append*}(\langle x, xs\rangle, \langle y, ys\rangle) & & \Leftarrow & \langle x, xs\rangle \approx e'_1, \ \langle y, ys\rangle \approx e'_2\\ \leadsto_{\{xs \mapsto y\}} & \langle x, ys\rangle & \Leftarrow & \langle x, y\rangle \approx e'_1, \langle y, ys\rangle \approx e'_2\\ \leadsto^*_{\sigma'} & \langle x, ys\rangle & \Leftarrow & \langle x, y\rangle \approx \langle d \!:\! z, z\rangle, \langle y, ys\rangle \approx \sigma'(e'_2)\\ \leadsto_{\{x \mapsto d:y, z \mapsto y\}} & \langle d \!:\! y, ys\rangle & \Leftarrow & \langle y, ys\rangle \approx \sigma'(e'_2)\end{array}$$

and the claim follows.

($\Leftarrow$) Consider the derivation

$$\begin{array}{lrcl}\texttt{append*}(\langle x, xs\rangle, \langle y, ys\rangle) & & \Leftarrow & \langle x, xs\rangle \approx e'_1, \langle y, ys\rangle \approx e'_2\\ \leadsto^*_{\sigma'} & \langle d \!:\! y, ys\rangle & \Leftarrow & \langle y, ys\rangle \approx \sigma'(e'_2)\end{array}$$

[2] Here we do not consider nested occurrences of `append`, although the proof scheme can be extended to cover this case by using an appropriate induction.

Since e'_1 is lazily regular, we have $e'_1 \leadsto_{\sigma'} \langle d{:}z, z\rangle$ in $\mathcal{R}_b$. Hence, by equivalence (*), $e_1 \leadsto_\sigma d{:}[]$ in $\mathcal{R}_a$, where $\sigma = \sigma'\ [\mathcal{V}ar(e_1)]$. Although the evaluation of e_1 and the calls to `append` are interleaved due to the laziness of `append`, we know that $\mathtt{append}(e_1, e_2) \leadsto^*_\sigma d{:}\sigma(e_2)$ by definition of needed narrowing, which completes the proof.

Note that requiring e'_1 to be lazily regular is not a real restriction in our context, since terminating functions fulfill this condition by the manner in which we introduce difference-lists in the base cases of recursive functions. On the other hand, if we were only interested in proving an equivalence w.r.t. head normal forms, we conjecture that this restriction could be safely dropped.

Now we concentrate on stages (d) and (e). Let $\mathcal{R}_c$ be the program obtained from stage (c) and $\mathcal{R}_e$ be the output of stage (e). In order to prove the correctness of this step, we prove that for each function symbol `f` in $\mathcal{R}_c$ (defined in terms of some `f*`) we have a semantically equivalent function `f` in $\mathcal{R}_e$ (defined in terms of $\mathtt{f}_{\mathtt{acc}}$). For the sake of simplicity, let us consider a recursive function of the form:

$$\begin{array}{l}
\mathtt{f}(\overline{x_n}) \ = y \ \Leftarrow \ \langle y, []\rangle \approx \mathtt{f*}(\overline{x_n}) \\
\mathtt{f*}(\overline{a_n}) = \langle d{:}y, y\rangle \\
\mathtt{f*}(\overline{b_n}) = \langle d'{:}z, y\rangle \ \Leftarrow \ \langle z, e{:}y\rangle \approx \mathtt{f*}(\overline{s_n})
\end{array}$$

in $\mathcal{R}_c$, where d, d', e represent a (possibly empty) sequence of elements of the form $d_1{:}\ldots{:}d_k$, $k \geq 0$. According to the transformation rules in stages (d) and (e), we produce the following definition:

$$\begin{array}{l}
\mathtt{f}(\overline{x_n}) \qquad = \mathtt{f}_{\mathtt{acc}}(\overline{x_n}, []) \\
\mathtt{f}_{\mathtt{acc}}(\overline{a_n}, y) = d{:}y \\
\mathtt{f}_{\mathtt{acc}}(\overline{b_n}, y) = d'{:}\mathtt{f}_{\mathtt{acc}}(\overline{s_n}, e{:}y)
\end{array}$$

in $\mathcal{R}_e$. Given a (finite) list $c_1{:}...{:}c_k{:}[]$, in order to prove that $\mathtt{f}(\overline{c_n}) \leadsto^*_\sigma c_1{:}...{:}c_k{:}[]$ in $\mathcal{R}_c$ iff $\mathtt{f}(\overline{c_n}) \leadsto^*_\sigma c_1{:}...{:}c_k{:}[]$ in $\mathcal{R}_e$, it suffices to prove that $\mathtt{f*}(\overline{c_n}) \leadsto^*_\sigma \langle c_1{:}...{:}c_k{:}y, y\rangle$ in $\mathcal{R}_c$ iff $\mathtt{f}_{\mathtt{acc}}(\overline{c_n}, []) \leadsto^*_\sigma c_1{:}...{:}c_k{:}[]$ in $\mathcal{R}_e$. To prove this claim by induction, we first generalize it as follows:

$$\begin{array}{l}
\langle t{:}z, y\rangle \ \Leftarrow \ \langle z, r{:}y\rangle \approx \mathtt{f*}(\overline{c_n}) \ \leadsto^*_\sigma \ \langle t'{:}l{:}r'{:}y, y\rangle \quad \text{in } \mathcal{R}_c \\
\quad \text{iff} \ \ t{:}\mathtt{f}_{\mathtt{acc}}(\overline{c_n}, r{:}[]) \ \leadsto^*_{\sigma'} \ t'{:}l{:}r'{:}[] \ \ \text{in } \mathcal{R}_e
\end{array}$$

where $\sigma = \sigma'\ [\mathcal{V}ar(\{\mathtt{f*}(\overline{c_n}), t, r\})]$, the expressions t, r, l represent (possibly empty) sequences of elements of the form $t_1{:}\ldots{:}t_k$, $k \geq 0$, and t', r' are constructor instances of t, r (in particular, $t' = \sigma(t), r' = \sigma(r)$). Now we proceed by induction on the length of the former derivation. The base case is immediate by applying the first rules of `f*` and $\mathtt{f}_{\mathtt{acc}}$, respectively. Let us consider the inductive case. By applying the second rule of `f*`, we have:

$$\langle t{:}z, y\rangle \Leftarrow \ \langle z, r{:}y\rangle \approx \mathtt{f*}(\overline{c_n}) \ \leadsto_\theta \ \langle t'{:}z, y\rangle \Leftarrow \ \langle z, r'{:}y\rangle \approx \langle d'{:}z', y'\rangle, \langle z', e{:}y'\rangle \approx \mathtt{f*}(\overline{s_n})$$

and, by unfolding the first equation in the condition:

$$\langle t'{:}d'{:}z', y\rangle \ \Leftarrow \ \langle z', e{:}r'{:}y\rangle \approx \mathtt{f*}(\overline{s_n})$$

On the other hand, by applying the second rule of $\mathtt{f}_{\mathtt{acc}}$ to $t{:}\mathtt{f}_{\mathtt{acc}}(\overline{c_n}, r{:}[])$, we have:

$$t{:}\mathtt{f}_{\mathtt{acc}}(\overline{c_n}, r{:}[]) \ \leadsto_{\theta'} \ t'{:}d'{:}\mathtt{f}_{\mathtt{acc}}(\overline{s_n}, e{:}r'{:}[])$$

where $\theta = \theta'\ [\mathcal{V}ar(\{\mathtt{f*}(\overline{c_n}), t, r\})]$. The claim follows by the inductive hypothesis.

$$
\begin{array}{lcl}
 & & \mathtt{f}(\overline{\mathtt{x_n}}) \quad\quad = \mathtt{f_{acc}}(\overline{\mathtt{x_n}}, []) \\
\mathtt{f}(\overline{\mathtt{s_n}}) = [] & \Longrightarrow & \mathtt{f_{acc}}(\overline{\mathtt{s_n}}, \mathtt{y}) = \mathtt{y} \\
\mathtt{f}(\overline{\mathtt{t_n}}) = \mathtt{m_1} : \mathtt{append}(\mathtt{f}(\overline{\mathtt{t'_n}}), \mathtt{m_2} : [])) & & \mathtt{f_{acc}}(\overline{\mathtt{t_n}}, \mathtt{y}) = \mathtt{m_1} : \mathtt{f_{acc}}(\overline{\mathtt{t'_n}}, \mathtt{m_2} : \mathtt{y}))
\end{array}
$$

$$
\begin{array}{lcl}
 & & \mathtt{f}(\overline{\mathtt{x_n}}) \quad\quad = \mathtt{f_{acc}}(\overline{\mathtt{x_n}}, []) \\
\mathtt{f}(\overline{\mathtt{s_n}}) = [] & \Longrightarrow & \mathtt{f_{acc}}(\overline{\mathtt{s_n}}, \mathtt{y}) = \mathtt{y} \\
\mathtt{f}(\overline{\mathtt{t_n}}) = \mathtt{m_1} : \mathtt{append}(\mathtt{f}(\overline{\mathtt{t'_n}}), \mathtt{m_2} : \mathtt{f}(\overline{\mathtt{t''_n}})) & & \mathtt{f_{acc}}(\overline{\mathtt{t_n}}, \mathtt{y}) = \mathtt{m_1} : \mathtt{f_{acc}}(\overline{\mathtt{t'_n}}, \mathtt{m_2} : \mathtt{f_{acc}}(\overline{\mathtt{t''_n}}, \mathtt{y}))
\end{array}
$$

$$
\begin{array}{l}
\mathtt{f}(\overline{\mathtt{s_n}}) = [] \\
f(\overline{t_n}) = \mathtt{m_1} : \mathtt{append}(\mathtt{append}(\mathtt{f}(\overline{\mathtt{t'_n}}), \mathtt{m_2} : \mathtt{f}(\overline{\mathtt{t''_n}})), \mathtt{m_3} : [])
\end{array}
$$

$$
\Longrightarrow \quad
\begin{array}{lcl}
\mathtt{f}(\overline{\mathtt{x_n}}) & = & \mathtt{f_{acc}}(\overline{\mathtt{x_n}}, []) \\
\mathtt{f_{acc}}(\overline{\mathtt{s_n}}, \mathtt{y}) & = & \mathtt{y} \\
\mathtt{f_{acc}}(\overline{\mathtt{t_n}}, \mathtt{y}) & = & \mathtt{m_1} : \mathtt{f_{acc}}(\overline{\mathtt{t'_n}}, \mathtt{m_2} : \mathtt{f_{acc}}(\overline{\mathtt{t''_n}}, \mathtt{m_3} : \mathtt{y}))
\end{array}
$$

where $\mathtt{m_1}, \mathtt{m_2}, \mathtt{m_3}$ are (possibly empty) sequences of the form $d_1 : d_2 : \ldots : d_k$, with $k \geq 0$.

Fig. 2. Matching scheme

4.4 Effectiveness of the Transformation

Throughout this section, our aim has been to define an automatic method for achieving the effect of the difference-list transformation over functional logic programs. We have not been concerned with the efficiency of its implementation. It turns out that some of the stages that we have introduced appear to be expensive to implement. Thus, for a first attempt of integrating the method into a real compiler, we have defined a matching scheme which is both simple and effective. For our transformation, we discovered that, in practice, many doubly recursive functions ensure a gain in efficiency from the transformation (also some single recursive functions, provided they use `append` to concatenate some elements to the result of the recursive call). These functions are matched by three simple, local transformation rules depicted in Fig. 2 and, thus, replaced by equivalent functions without calls to `append`.

As an example, we consider the towers of Hanoi:

$$
\begin{array}{ll}
\mathtt{hanoi}(\mathtt{0}, \mathtt{a}, \mathtt{b}, \mathtt{c}) & = [] \\
\mathtt{hanoi}(\mathtt{S}(\mathtt{n}), \mathtt{a}, \mathtt{b}, \mathtt{c}) & = \mathtt{append}(\mathtt{hanoi}(\mathtt{n}, \mathtt{a}, \mathtt{c}, \mathtt{b}), (\mathtt{a}, \mathtt{b}) : \mathtt{hanoi}(\mathtt{n}, \mathtt{c}, \mathtt{b}, \mathtt{a}))
\end{array}
$$

where the first argument is of type $\mathtt{Nat} \ = \ \mathtt{0} \mid \mathtt{S}(\mathtt{Nat})$; $\mathtt{a}$, $\mathtt{b}$ and $\mathtt{c}$ represent the three towers, and $(\mathtt{a}, \mathtt{b})$ a movement of a plate from $\mathtt{a}$ to $\mathtt{b}$. By considering that m_1 is an empty sequence and $m_2 = (\mathtt{a}, \mathtt{b})$, the second rule of the scheme matches and transforms the program into the following optimized version without concatenations:

$$
\begin{array}{ll}
\mathtt{hanoi}(\mathtt{n}, \mathtt{a}, \mathtt{b}, \mathtt{c}) & = \mathtt{han}(\mathtt{n}, \mathtt{a}, \mathtt{b}, \mathtt{c}, []) \\
\mathtt{han}(\mathtt{0}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{y}) & = \mathtt{y} \\
\mathtt{han}(\mathtt{S}(\mathtt{n}), \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{y}) & = \mathtt{han}(\mathtt{n}, \mathtt{a}, \mathtt{c}, \mathtt{b}, (\mathtt{a}, \mathtt{b}) : \mathtt{han}(\mathtt{n}, \mathtt{c}, \mathtt{b}, \mathtt{a}, \mathtt{y}))
\end{array}
$$

where all the concatenations have actually disappeared.

5 Experimental Evaluation

In order to evaluate experimentally our transformation, we have incorporated the optimization based on the matching scheme of Fig. 2 into the PACS compiler for Curry

[8] as an automatic source-to-source transformation which is transparent to the user. The language Curry is an international initiative to provide a common platform for the research, teaching and application of integrated functional logic languages [7]. To implement the optimization, we have used the standard intermediate representation of Curry programs: FlatCurry [8].[3]

To show the usefulness of our approach, we considered programs which are used in the literature to illustrate the benefits of difference-lists in Prolog (adapted to a functional logic syntax). The complete code of the benchmarks and a detailed description of the implementation can be found in [1]. The following table shows the performances of original programs (Original) w.r.t. the improved versions (Optimized) by the introduction of accumulating parameters:

Benchmarks	Original	Optimized	Speedup
$\texttt{rev}_{2000}$	3470	65	53.38
$\texttt{qsort}_{2000}$	1010	850	1.18
$\texttt{pre-order}_{2000}$	104	17	6.11
$\texttt{in-order}_{2000}$	105	16	6.56
$\texttt{post-order}_{2000}$	132	16	8.25
$\texttt{hanoi}_{17}$	4100	2160	1.89

Times are expressed in milliseconds and are the average of 10 executions. Runtime input goals were chosen to give a reasonably long overall time. In particular, goal subindices show the number of elements in the input lists or trees. Column Speedup shows the relative improvements achieved by the transformation, obtained as the ratio Original $\div$ Optimized, respectively. Results are encouraging, achieving significant speedups for some of the examples.

6 Related Work

The development of list-processing optimizations has been an active research topic both in functional and logic programming for the last decades. A related approach to difference-lists appeared early in [10], where Hughes introduced an optimized representation of lists, the so-called *abstract lists*, which are specially defined for a fast concatenation in functional programming. The idea behind their use is similar to that of logic difference-lists, although they are formulated in a different way. As opposite to our approach, the objective of [10] is not to provide an automatic algorithm to replace standard lists by abstract lists, but to introduce an efficient data structure to be used by the programmer. The idea of optimizing concatenations was taken one step forward by Wadler in [14], where he described local transformations for removing some concatenations from a program. The formalization of our stepwise process to introduce accumulating parameters is, apparently, not related with Wadler's transformation. Nevertheless, we strongly believe that over many examples both approaches produce a similar effect. A formal comparison between them could be useful. For instance, we think that our marking algorithm could be used within Wadler's technique

[3] A prototype implementation, together with some examples and documentation of the system is publicly available at: `http://www.dsic.upv.es/users/elp/soft.html`.

to identify those functions from which concatenations will be successfully removed. On the other hand, we could benefit from the simplicity of Wadler's rules in some steps of our transformation.

7 Conclusions and Future Work

We have presented a novel transformation for improving list-processing functions in the context of a multi-paradigm functional logic language: an automatic transformation based on the introduction of accumulating parameters. It has been shown practical and effective by testing it within a real functional logic compiler, the PACS compiler for Curry [8].

A promising direction for future work is the generalization of our stepwise transformation to arbitrary (algebraic) data types. Another interesting topic is the definition of abstract measures to quantify the performance of functional logic programs, i.e., measures independent of concrete implementations. We expect that these measures also shed some light to find new optimizations and to determine their power.

References

1. E. Albert, C. Ferri, F. Steiner, and G. Vidal. List-Processing Optimizations in a Multi-Paradigm Declarative Language. Technical Report DSIC, UPV, 2000. Available from URL: `http://www.dsic.upv.es/users/elp/papers.html`.
2. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In A. Middeldorp and T. Sato, editors, *Proc. of FLOPS'99*, pages 147–162. Springer LNCS 1722, 1999.
3. S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Princ. of Prog. Languages, Portland*, pages 268–279, 1994.
5. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.
6. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
7. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
8. M. Hanus (ed.), S. Antoy, J. Koj, R. Sadre, and F. Steiner. PACS 1.0: User Manual. Technical report, RWTH Aachen, Germany, 1999.
9. G. Huet and J.J. Lévy. Computations in orthogonal rewriting systems, Part I + II. In J.L. Lassez and G.D. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–443, 1992.
10. J. Hughes. A Novel representation of Lists and its Application to the Function `reverse`. Technical Report PMG-38, Programming Methodology Group, Department of Computer Science, Chalmers Institute of Technology, Sweden, 1984.
11. K. Marriott and H. Søndergaard. Difference-list Transformation for Prolog. *New Generation Computing*, 11(2):125–157, October 1993.

12. L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.
13. H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In *Proc. of Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.
14. P.L. Wadler. The Concatenate Vanishes. Technical report, Department of Computing Science, University of Glasgow, UK, 1987.