A Difference-List Transformation for Functional Logic Languages*

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,
{ealbert,cferri,gvidal}@dsic.upv.es
[2] Institut für Informatik, Christian-Albrechts-Universität Kiel,
Olshausenstr. 40, D-24098 Kiel, Germany, fst@informatik.uni-kiel.de

Abstract. The multi-paradigm language Curry integrates features from functional, logic, and concurrent programming. Within a multi-paradigm framework, one would expect that optimizations originally developed for the pure paradigms were still effective and correct. In this work, we consider a well-known list-processing optimization from logic programming, i.e., *difference-lists*, and study its adaptation to our integrated setting. Unfortunately, the use of difference-lists in Curry is impractical due to the absence of non-strict equality in the language. Despite all, we have developed a novel transformation which achieves a similar effect over functional logic programs.

1 Introduction

Curry is a universal programming language aiming at the amalgamation of the most important declarative programming paradigms, namely functional programming and logic programming. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). A detailed description of the features of Curry, its syntax and operational semantics can be found in [9].

In functional (logic) programming, it is common to define functions by composing calls to small functions such that, in order to transfer results among calls, intermediate lists are often used. This usually gives rise to inefficient programs. Thus, there have been many in-depth discussions on how to optimize these compositional programs. The most popular such transformations for functional languages are Wadler's deforestation [15] and short cut deforestation [7], which have successfully been adapted to Curry programs [3, 2].

In this paper we want to study a list-processing optimization from the logic programming community. Most Prolog programmers know how to use *difference-*

* This work has been partially supported by CICYT TIC 98-0445-C03-01, by Acción Integrada hispano-alemana HA1997-0073, and by the German Research Council (DFG) under grant Ha 2457/1-1.

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$1:2:xs, xs$\rangle$, where xs is a logical variable. The key to succeed in optimizing programs by difference-lists is the use of a constant-time concatenation: append($\langle$x, y$\rangle$, $\langle$y, z$\rangle$, $\langle$x, 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: append$*$ $\langle$x, y$\rangle$ $\langle$y, z$\rangle$ = $\langle$x, z$\rangle$ and, consequently, prevents us from having difference-lists in Curry (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" [13]. 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.

After some preliminary definitions in the next section, we describe the development of a fully automatic list-processing optimization technique based on the introduction of accumulating parameters in Section 3. An experimental evaluation of this optimization is shown in Section 4. Section 5 presents related work and Section 6 contains our conclusions. An extended version of this paper with more details and missing proofs can be found in [1].

2 Preliminaries

We assume familiarity with basic notions from term rewriting [6] and functional logic programming [8]. In this work we consider a (*many-sorted*) *signature* Σ partitioned into a set $\mathcal{C}$ of *constructors* and a set $\mathcal{F}$ of *defined* functions. 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 is *linear* if it does not contain multiple occurrences of one variable. 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. 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 identity substitution is denoted by id.

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. In the remainder of this paper, a functional logic *program* is a left-linear CB-TRS. A *rewrite step* is an application of a rewrite rule to a term,

i.e., $t \to_{p,R} s$ if there exists a position p in t, a rewrite rule $R = (l = r)$ and a substitution σ with $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$.

The operational semantics of integrated languages is usually based on *narrowing*, a combination of variable instantiation and reduction. Formally, $s \leadsto_{p,R,\sigma} t$ is a *narrowing step* if p is a non-variable position in s and $\sigma(s) \to_{p,R} t$. 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 $\sigma = id$).

The Language Curry. The kernel of the multi-paradigm language Curry is based on *needed narrowing* and *inductively sequential* programs (see [4] for a precise definition). 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. The following example shows the essence of the evaluation by needed narrowing.

Example 1. Consider the data type `List a` defined as

```
data List a = [] | a : List a
```

and the function `isShorter :: List a → List a → Bool`, which is defined by the following equations:

```
isShorter []     ys     = True
isShorter (x:xs) []     = False
isShorter (x:xs) (y:ys) = isShorter xs ys
```

A possible evaluation of the expression `isShorter (x:xs) z` can be obtained by instantiating `z` to `(y:ys)` to apply the third equation, followed by the instantiation of `xs` to `[]` to apply the first equation:

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

3 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* [13]. The development is inspired by the well-known difference-list transformation from the logic programming community [12, 13]. The idea behind the difference-list transformation of [12] 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. The key to succeed in optimizing programs by difference-lists is the use of a constant-time append: `append`$(\langle \texttt{x}, \texttt{y}\rangle, \langle \texttt{y}, \texttt{z}\rangle, \langle \texttt{x}, \texttt{z}\rangle)$.

In an attempt to adapt this technique to a functional logic context, we find several problems. As discussed in the introduction, we cannot encode the concatenation of difference-lists by means of the rule: $\texttt{append*}\ \langle \texttt{x}, \texttt{y}\rangle\ \langle \texttt{y}, \texttt{z}\rangle = \langle \texttt{x}, \texttt{z}\rangle$, since it is not left-linear. More generally, the manipulation of difference-lists requires the use of non-strict equality in order to *assign* terms to the *pointers* (i.e., suffixes) of difference-lists. Since only strict equality is allowed in Curry, we are interested in a transformation process in which the final program does not contain occurrences of difference-lists (nor calls to `append*`).

An Intuitive Example. Now, we illustrate intuitively how our method proceeds. Let us consider the following function to reverse the elements of a list:

$$\begin{array}{l}\texttt{reverse}\ [] \quad\ = [] \\ \texttt{reverse}\ (\texttt{x} : \texttt{xs}) = \texttt{append}\ (\texttt{reverse}\ \texttt{xs})\ (\texttt{x} : [])\end{array}$$

Following essentially the method of [12], we have to replace the arguments of the function `append` and the *result* of both functions by difference-lists. From the work in [12], we know how to transform different kinds of standard lists into difference-lists (e.g., list `[]` is transformed into difference-list $\langle \texttt{x}, \texttt{x}\rangle$, list $(\texttt{x} : [])$ into $\langle \texttt{x} : \texttt{y}, \texttt{y}\rangle$, etc.); now, due to the functional syntax of Curry, we are faced with a new situation: how can we transform a term rooted by a defined function into a difference-list? To solve this problem, we allow the *flattening* of some calls by using a sort of conditional expressions. Here, we use a syntactic (non-strict) equality "$\approx$" for the equations in the condition (although they will not appear in the final program). In this way, we get the following transformed program:

$$\begin{array}{l}\texttt{rev*}\ [] \quad\ = \langle \texttt{x}, \texttt{x}\rangle \\ \texttt{rev*}\ (\texttt{x} : \texttt{xs}) = \texttt{append*}\ \langle \texttt{z}, \texttt{zs}\rangle\ \langle \texttt{x} : \texttt{ys}, \texttt{ys}\rangle\ \Leftarrow \langle \texttt{z}, \texttt{zs}\rangle \approx \texttt{rev*}\ \texttt{xs}\end{array}$$

We also add a new equation to connect the old `reverse` with the difference-list version:

$$\texttt{reverse}\ \texttt{xs} = \texttt{y}\ \Leftarrow \langle \texttt{y}, []\rangle \approx \texttt{rev*}\ \texttt{xs}$$

Similarly to the case of logic programming, the constant-time `append*` is defined by the rule:

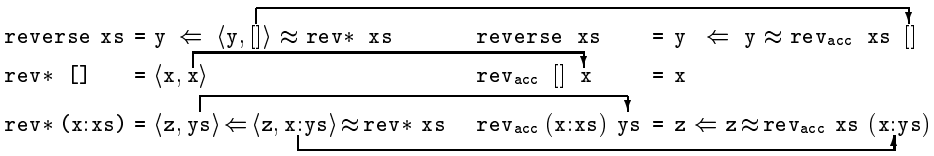
$$\texttt{append*}\ \langle \texttt{x}, \texttt{y}\rangle\ \langle \texttt{y}, \texttt{z}\rangle\ =\ \langle \texttt{x}, \texttt{z}\rangle$$

We only use this rule during the transformation, but it will not be included in the final program. According to the method of [12], the next stage is to unfold the calls to `append*`:

$$\begin{array}{l}\texttt{reverse}\ \texttt{xs}\ = \texttt{y}\ \Leftarrow \langle \texttt{y}, []\rangle \approx \texttt{rev*}\ \texttt{xs} \\ \texttt{rev*}\ [] \quad\ = \langle \texttt{x}, \texttt{x}\rangle \\ \texttt{rev*}\ (\texttt{x} : \texttt{xs}) = \langle \texttt{z}, \texttt{ys}\rangle\ \Leftarrow \langle \texttt{z}, \texttt{x} : \texttt{ys}\rangle \approx \texttt{rev*}\ \texttt{xs}\end{array}$$

Unlike the original transformation, now we want to remove difference-lists (and non-strict equations) from the program. The key 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 call to `rev*` is flattened using a conditional expression, we also need to move the second

argument of the difference-list to the corresponding call to `rev*`:



where `rev*` is renamed as $\texttt{rev}_{\texttt{acc}}$. By simplifying the equations in the conditions, we get:

$$\begin{array}{lll} \texttt{reverse xs} = \texttt{rev}_{\texttt{acc}}\ \texttt{xs}\ [] & \texttt{rev}_{\texttt{acc}}\ []\ \texttt{x} & = \texttt{x} \\ & \texttt{rev}_{\texttt{acc}}\ (\texttt{x} : \texttt{xs})\ \texttt{ys} & = \texttt{rev}_{\texttt{acc}}\ \texttt{xs}\ (\texttt{x} : \texttt{ys}) \end{array}$$

Here, thanks to the use of accumulating parameters, we avoid the concatenation of intermediate lists. In general, this optimization is able to produce superlinear speedups [12, 13].

The Method. Now we present a stepwise transformation to systematically introduce accumulating parameters. This way, we achieve the elimination of calls to `append` (similarly, in spirit, to the difference-list transformation of [12]).

(a) Marking Algorithm:

First, given a function to be optimized (whose result type is a list), 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 `f` (to be optimized)
2. Initialization: $\mathcal{M}_0 = \{\texttt{f}\}$, $i = 0$
3. Repeat
   - for each function in $\mathcal{M}_i$, mark the right-hand sides of the rules defining `f` in $\mathcal{R}$
   - propagate marks among expressions by applying the following rules:

     $\underline{\texttt{append}(t_1, t_2)} \to \texttt{append}(\underline{t_1}, \underline{t_2})$
     $\underline{t_1 : t_2} \to t_1 : \underline{t_2}$
     $\underline{\texttt{g}(t_1, \ldots, t_n)} \to \underline{\texttt{g}}(t_1, \ldots, t_n)$ where $\texttt{g} \in \mathcal{F}$ is different from `append`.

   - if there exists a marked expression $\underline{t}$ such that t is a variable, then return FAIL;
     else $\mathcal{M}_{i+1} = \{\texttt{h} \mid \underline{\texttt{h}}(t_1, \ldots, t_k)$ 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:

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

where y, y' are fresh variables not appearing in the program. Furthermore, we consider that all marked function symbols $\mathtt{f}$ in the resulting program are replaced by $\mathtt{f}*$. For instance, a rule of the form $\mathtt{f}(\mathtt{c}_1,\ldots,\mathtt{c}_n) = \mathtt{t}_1 : \mathtt{t}_2 : \underline{\mathtt{f}}(\mathtt{s}_1,\ldots,\mathtt{s}_n)$ is transformed into:

$$\mathtt{f}*(\mathtt{c}_1,\ldots,\mathtt{c}_n) = \langle \mathtt{t}_1 : \mathtt{t}_2 : \mathtt{y}, \mathtt{y}' \rangle \Leftarrow \langle \mathtt{y}, \mathtt{y}' \rangle \approx \mathtt{f}*(\mathtt{s}_1,\ldots,\mathtt{s}_n)$$

Similarly, those occurrences of `append` whose arguments have been replaced by difference-lists are renamed as `append`$*$. Note that, when the transformation of several terms gives rise to conditional expressions, all equations are joined into a single condition. Also, the following equation replaces the original definition of $\mathtt{f}$:

$$\mathtt{f}(x_1,\ldots,x_n) \;=\; y \;\Leftarrow \langle y, [] \rangle \approx \mathtt{f}*(x_1,\ldots,x_n)$$

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 (by a narrowing step) the calls to `append`$*$ using the following rule:

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

In this way, all calls to `append`$*$ disappear from the program.

(d) Use of Accumulating Parameters:

Then, we move the second argument of difference-lists to the corresponding function call:

$$\begin{array}{lcl} [\mathtt{f}*(t_1,\ldots,t_n) = \langle y, y' \rangle \Leftarrow C] & \rightarrow & [\mathtt{f}_{\mathtt{acc}}(t_1,\ldots,t_n,y') = y \Leftarrow C] \\ [t \Leftarrow \langle s, s' \rangle \approx \mathtt{f}*(t,\ldots,t_n)] & \rightarrow & [t \Leftarrow s \approx \mathtt{f}_{\mathtt{acc}}(t_1,\ldots,t_n,s')] \end{array}$$

This stage 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 left-hand sides of the equations in the conditions are always free variables).

We illustrate how our strategy proceeds with two examples. First, let us consider the following program, and the corresponding marked program (starting with function $\mathtt{f}$)[1]:

[1] Due to space limitations we write `app` instead of `append` and $\mathtt{f}_{\mathtt{a}}$ instead of $\mathtt{f}_{\mathtt{acc}}$.

$$
\begin{array}{lll}
\texttt{f [] y} & \texttt{= y : []} & \rightarrow \quad \texttt{f [] y} \quad \texttt{= y } \underline{\texttt{:}} \texttt{ []} \\
\texttt{f (x:xs) y} & \texttt{= app (f xs y) (x:(g xs))} & \rightarrow \quad \texttt{f (x:xs) y} = \texttt{app (}\underline{\texttt{f}}\texttt{ xs y) (x } \underline{\texttt{:}} \texttt{ (}\underline{\texttt{g}}\texttt{ xs))} \\
\texttt{g []} & \texttt{= []} & \rightarrow \quad \texttt{g []} \quad \texttt{= } \underline{\texttt{[]}} \\
\texttt{g (x:xs)} & \texttt{= x : (g xs)} & \rightarrow \quad \texttt{g (x:xs)} \quad \texttt{= x } \underline{\texttt{:}} \texttt{ (}\underline{\texttt{g}}\texttt{ xs)}
\end{array}
$$

After replacing the marked expressions by difference-lists, we get:

$$
\begin{array}{l}
\texttt{f x y} \quad = \texttt{z} \Leftarrow \langle \texttt{z}, [] \rangle \approx (\texttt{f*\ x y}) \\
\texttt{f* [] y} \quad = \langle \texttt{y:z}, \texttt{z} \rangle \\
\texttt{f* (x:xs) y} = \texttt{append*} \ \langle \texttt{z}, \texttt{z}' \rangle \ \ \langle \texttt{x:w}, \texttt{w}' \rangle \ \Leftarrow \ \langle \texttt{z}, \texttt{z}' \rangle \approx (\texttt{f* xs y}), \quad \langle \texttt{w}, \texttt{w}' \rangle \approx (\texttt{g* xs}) \\
\texttt{g x} \quad = \texttt{y} \ \Leftarrow \ \langle \texttt{y}, [] \rangle \approx (\texttt{g* x}) \\
\texttt{g* []} \quad = \langle \texttt{y}, \texttt{y} \rangle \\
\texttt{g* (x:xs)} \quad = \langle \texttt{x:y}, \texttt{y}' \rangle \ \Leftarrow \ \langle \texttt{y}, \texttt{y}' \rangle \approx (\texttt{g* xs})
\end{array}
$$

By unfolding the call to `append*`, the third rule is transformed into:

$$\texttt{f* (x:xs) y} = \langle \texttt{z}, \texttt{w}' \rangle \ \Leftarrow \ \langle \texttt{z}, \texttt{x:w} \rangle \approx (\texttt{f* xs y}), \quad \langle \texttt{w}, \texttt{w}' \rangle \approx (\texttt{g* xs})$$

By introducing accumulating parameters (and, then, simplifying the conditions), we get:

$$
\begin{array}{lll}
\texttt{f x y} & \texttt{= z} \Leftarrow \texttt{z} \approx (\texttt{f}_\texttt{a}\texttt{ x y []}) & \rightarrow \quad \texttt{f x y} \quad = \texttt{f}_\texttt{a}\texttt{ x y []} \\
\texttt{f}_\texttt{a}\texttt{ [] y z} & \texttt{= y : z} & \rightarrow \quad \texttt{f}_\texttt{a}\texttt{ [] y z} \quad = \texttt{y : z} \\
\texttt{f}_\texttt{a}\texttt{ (x:xs) y w}' & \texttt{= z} \Leftarrow \texttt{z} \approx (\texttt{f}_\texttt{a}\texttt{ xs y (x:w)}, & \rightarrow \quad \texttt{f}_\texttt{a}\texttt{ (x:xs) y w}' = \texttt{f}_\texttt{a}\texttt{ xs y (x:(g}_\texttt{a}\texttt{ xs w}'\texttt{))} \\
 & \qquad \texttt{w} \approx (\texttt{g}_\texttt{a}\texttt{ xs w}') & \\
\texttt{g x} & \texttt{= y} \Leftarrow \texttt{y} \approx (\texttt{g}_\texttt{a}\texttt{ x []}) & \rightarrow \quad \texttt{g x} \quad = (\texttt{g}_\texttt{a}\texttt{ x []}) \\
\texttt{g}_\texttt{a}\texttt{ [] y} & \texttt{= y} & \rightarrow \quad \texttt{g}_\texttt{a}\texttt{ [] y} \quad = \texttt{y} \\
\texttt{g}_\texttt{a}\texttt{ (x:xs) y}' & \texttt{= x:y} \Leftarrow \texttt{y} \approx (\texttt{g}_\texttt{a}\texttt{ xs y}') & \rightarrow \quad \texttt{g}_\texttt{a}\texttt{ (x:xs) y}' \quad = \texttt{x:(g}_\texttt{a}\texttt{ xs y}')
\end{array}
$$

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 function:

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

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

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

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 different lists of the arguments of `dapp` and, thus, we would produce a non-legal Curry 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). 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

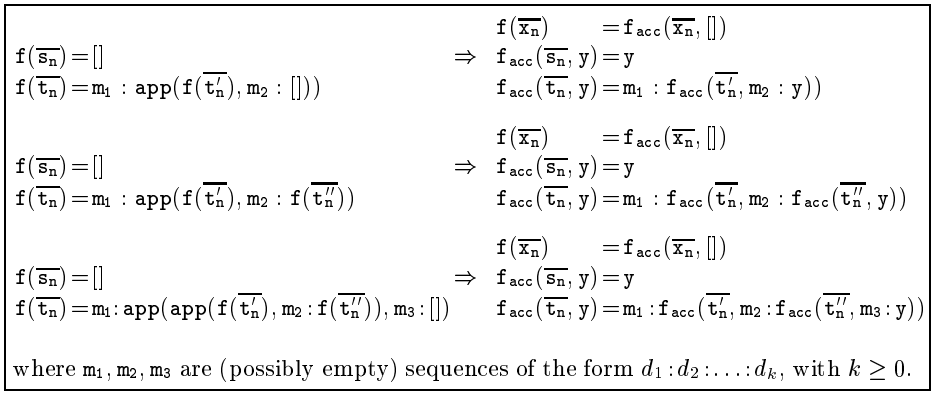


Fig. 1. Matching scheme

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 Experimental Evaluation

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 in a real compiler, we have defined a matching scheme which is both simple and effective. In general, scheme-based transformations identify known patterns of functions in programs and optimize them by using fixed rules about these known patterns. 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 Figure 1 (again we use `app` as abbreviation for `append`) and, thus, replaced by equivalent functions without calls to `append`. The advantage is that implementation is cheap and guarantees improvement in efficiency for the functions which match the scheme.

As an example, we consider the towers of Hanoi:

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

where the first argument is of type $\mathtt{Nat} = 0 \mid \mathtt{S}(\mathtt{Nat})$; `a`, `b` and `c` represent the three towers, and $(\mathtt{a}, \mathtt{b})$ a movement of a plate from `a` to `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:

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

A prototype implementation of the matching scheme, together with some examples and documentation of the system, is publicly available at:

`http://www.dsic.upv.es/users/elp/soft.html`.

Our prototype system optimizes Curry programs and is completely written in Curry itself. We have incorporated the optimizations into the PAKCS compiler for Curry as an automatic source-to-source transformation which is transparent to the user. The prototype system was successfully used to optimize several examples typical of the difference-list transformation, like the reverse function, the quicksort algorithm or functions for collecting the leaves of a tree by traversing it in pre-, in- or post-order. The complete code of all benchmarks and a detailed description of the implementation can be found in [1]. The following table compares the performances of original programs (Original) w.r.t. the improved versions (Optimized).

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 subindexes show the number of elements in the input lists or trees. Columns Speedup show the relative improvements achieved by the transformations, obtained as the ratio Original $\div$ Optimized. Results are encouraging, achieving significant speedups for some of the examples.

5 Related Work

A related approach to difference-lists appeared early in [11], 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 [11] 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 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.

6 Conclusions and Future Work

We have presented an automatic transformation based on the introduction of accumulating parameters. The transformations has been shown practical and effective by testing it within a real functional logic compiler, the PAKCS compiler for Curry [10].

An interesting topic for further research is the extension of the proposed techniques to arbitrary algebraic data types. We also plan to consider the integration of short cut deforestation [2] and the difference-list transformation presented in this paper into a single transformation. We believe that this could be a promising approach to additionally achieve the effect of an automatic and cheap *tupling* transformation similar to Chin's tupling [5], another well-known technique to improve list-processing functions, whose aim is essentially to remove multiple traversals of the same data structure.

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. E. Albert, C. Ferri, F. Steiner, and G. Vidal. List-Processing Optimizations in Curry. In *Proceedings of the 9th Intl. Workshop on Functional and Logic Programming*, 2000. To appear.
3. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A Transformation System for Lazy Functional Logic Programs. In *Proc. of FLOPS'99*, pages 147–162. Springer LNCS 1722, 1999.
4. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, 1994.
5. W. Chin. Towards an Automated Tupling Strategy. In *Proc. of PEPM'93*, pages 119–132. ACM, New York, 1993.
6. 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.
7. A.J. Gill, J. Launchbury, and S.L. Peyton Jones. A Short Cut to Deforestation. In *Proc. of FPLCA'93*, pages 223–232, New York, NY (USA), 1993. ACM Press.
8. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

9. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at: `http://www.informatik.uni-kiel.de/~curry/`, 2000.
10. M. Hanus (ed.), S. Antoy, J. Koj, R. Sadre, and F. Steiner. PAKCS 1.2: User Manual. Available at: `http://www.informatik.uni-kiel.de/~curry/`, 2000.
11. 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.
12. K. Marriott and H. Søndergaard. Difference-list Transformation for Prolog. *New Generation Computing*, 11(2):125–157, October 1993.
13. L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.
14. P.L. Wadler. The Concatenate Vanishes. Technical report, Department of Computing Science, University of Glasgow, UK, 1987.
15. P.L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.