A Difference-List Transformation for Functional Logic Languages

Frank Steiner

CAU Kiel, Germany

Joint work with

Elvira Albert, Cèsar Ferri and Germán Vidal

Universidad Politécnica de Valencia, Spain

Introduction

Functional/Logic languages: lists as main data structure:

+ easy to handle
+ good intermediate structure for connection different functions/program parts

- intermediate structures consume memory and time
- often several traversals of the same structure

```
rev []     = []
rev (x:xs) = append (rev xs) [x]

rev [1,2,3,4,5]
⇝... append (append (rev [3,4,5]) [2]) [1]
⇝... append (append  [5,4,3]      [2]) [1]
```

Transformations to avoid/remove (repeated traversal of) intermediate structures, for instance:

- functional: short cut deforestation
- logic: difference-lists

Curry: A Truly Integrated Functional Logic Language

[Dagstuhl'96, POPL'97]

- multi-paradigm language, combines
  - functional programming
  - logic programming
  - concurrent programming
- based on an optimal evaluation strategy (residuation + narrowing)
- features like nondeterministic search, monadic I/O, constraints, polymorphic type system, communication and distributed programming
- conservative extension of lazy functional and (concurrent) logic programming
  - try to benefit from both sides
  - known transformations adaptable?
    - short-cut deforestation: done
    - difference-list transformation: similarly

Difference-Lists

Idea: Represent a list as difference of two other lists:

[1,2,3] is the difference of [1,2,3,4,5] and [4,5]
or [1,2,3,8] and [8]

Difference-lists in Prolog: <[1,2,3|Xs],Xs>

- List plus pointer to its tail: $\boxed{1} \longrightarrow \boxed{2} \longrightarrow \boxed{3} \longrightarrow$ nil
- empty list: <Xs,Xs>
- concatenation in constant time:

```
append*(<A,B>,<B,C>,<A,C>).

?- append*(<[1,2,3|Xs],Xs>,<[4,5,6|Ys],Ys>,<Z,Zs>)

⇝ {Xs ↦ [4,5,6|Ys]} ⇒ <Z,Zs> = <[1,2,3,4,5,6|Ys],Ys>
```

`reverse` with Difference-Lists

- Normal definition in Prolog:

```
rev([],[]).
rev([X|Xs],Ys) :- rev(Xs,Zs), append (Zs,[X],Ys).
```

- Difference-List version:

```
rev*([],<Ys,Ys>).
rev*([X|Xs],<Y,Ys>) :- rev*(Xs,<Z,Zs>),
                                         append*(<Z,Zs>,<[X|T],T>,<Y,Ys>)
```

- Unfold `append`*:

```
rev*([],<Ys,Ys>).
rev*([X|Xs],<Y,Ys>) :- rev*(Xs,<Y,[X|Ys]>).
```

- Same structure as accumulating version:

```
rev([],   Ys,Ys).
rev([X|Xs],   Y,Ys)   :-   rev(Xs,Y,[X|Ys]).
```

Difference-Lists in Curry

- Difference-Lists are not directly adaptable to Curry:

  1. `append <a,b> <b,c> = <a,c>`

     ⇒ non-linear left-hand side

  2. `append <a,b> <d,c> | b=:=d = <a,c>`

     ⇒ different operational behaviour

- Idea: Transform classical version

  ```
  rev []     = []
  rev (x:xs) = append (rev xs) [x]
  ```

  into accumulating version

  ```
  rev xs = rev_acc xs []

  rev_acc []     ys = ys
  rev_acc (x:xs) ys = rev_acc xs (x:ys)
  ```

  by using difference-lists as intermediate stage.

Difference-List Transformation for Curry

```
rev []     = []
rev (x:xs) = append (rev xs) [x]

append <a,b> <b,c> = <a,c>
```

First step: Mark transformable expressions

Second step: Make functions calculate difference-lists

```
rev1 []     = <ys,ys>
rev1 (x:xs) = append <y,ys> <(x:zs),zs>
                ⇐ <y,ys> ≈ rev1 xs
```

"$\approx$" denotes syntactic (non-strict) equality

Third step: unfold append

```
rev2 []     = <ys,ys>
rev2 (x:xs) = <y,zs>
                ⇐ <y,(x:zs)> ≈ rev2 xs
```

Difference-List Transformation for Curry

`append <a,b> <b,c> = <a,c>`

Example: $\texttt{rev}_1\ \texttt{[1,2,3,4]}$

$$
\begin{array}{lll}
\leadsto & \texttt{append <y,ys>}\qquad\qquad\ \ \texttt{<(1:zs),zs>} & \Leftarrow\ \texttt{<y,ys>} \approx \texttt{rev}_1\ \texttt{[2,3,4]} \\
\overset{*}{\leadsto} & \texttt{append <y,ys>}\qquad\qquad\ \ \texttt{<(1:zs),zs>} & \Leftarrow\ \texttt{<y,ys>} \approx \texttt{<(4:3:2:as),as>} \\
\overset{*}{\leadsto} & \texttt{append <(4:3:2:as),as> <(1:zs),zs>} & \\
\overset{\{\texttt{as}\mapsto\texttt{1:zs}\}}{\leadsto} & \qquad\texttt{<(4:3:2:1:zs),zs>} &
\end{array}
$$

Unfolded version:

$$
\begin{array}{lll}
\texttt{rev}_2\ \texttt{[1,2,3,4]} & \leadsto & \texttt{<y,zs>} \Leftarrow \texttt{<y,(1:zs)>} \approx \texttt{rev}_2\ \texttt{[2,3,4]} \\
 & \overset{*}{\leadsto} & \texttt{<y,zs>} \Leftarrow \texttt{<y,(1:zs)>} \approx \texttt{<(4:3:2:as),as>} \\
 & & \quad\Rightarrow\ \texttt{as} \mapsto \texttt{1:zs} \quad \Rightarrow\ \texttt{y} \mapsto \texttt{4:3:2:as} = \texttt{4:3:2:1:zs} \\
 & \leadsto & \texttt{<(4:3:2:1:zs),zs>}
\end{array}
$$

Difference-List Transformation for Curry

Fourth step: transfer difference-list tail into function parameter

$$\begin{array}{l}
\texttt{rev}_2\ \texttt{[]}\qquad\quad\ = \langle\texttt{ys},\texttt{ys}\rangle \\
\texttt{rev}_2\ \texttt{(x:xs)} = \langle\texttt{y},\texttt{zs}\rangle \Leftarrow \langle\texttt{y},\texttt{(x:zs)}\rangle \approx \texttt{rev}_2\ \texttt{xs}
\end{array}$$

$$\Downarrow$$

$$\begin{array}{l}
\texttt{rev}_{acc}\ \texttt{[]}\qquad\quad\ \texttt{ys} = \texttt{ys} \\
\texttt{rev}_{acc}\ \texttt{(x:xs)}\ \texttt{zs} = \texttt{ys} \Leftarrow \texttt{ys} \approx \texttt{rev}_{acc}\ \texttt{xs}\ \texttt{(x:zs)}
\end{array}$$

Direction of list construction reversed!

Fifth step: Simplify and add connecting definition

$$\begin{array}{l}
\texttt{rev xs} = \texttt{rev}_{acc}\ \texttt{xs}\ \texttt{[]} \\
\\
\texttt{rev}_{acc}\ \texttt{[]}\qquad\quad\ \texttt{ys} = \texttt{ys} \\
\texttt{rev}_{acc}\ \texttt{(x:xs)}\ \texttt{zs} = \texttt{rev}_{acc}\ \texttt{xs}\ \ \texttt{(x:zs)}
\end{array}$$

Correctness

5-step transformation algorithm:

1. Marking
2. Introduction of difference-lists
3. Unfolding of append
4. Creation of accumulating parameters
5. Simplification

Correctness results

1. Algorithm terminates with `"fail"` or transformed programm.
2. Transformed programs are valid Curry programs.
3. `f` and $\mathtt{f}_{acc}$ are equivalent w.r.t. finite reductions to finite lists:

$$
\begin{array}{ccc}
f\ e_1 \ldots e_n & \overset{i}{\leadsto} & \mathtt{[}l_1, \ldots, l_k\mathtt{]} \\
 & iff & \\
f_{acc}\ e_1 \ldots e_n\ \mathtt{[]} & \overset{j}{\leadsto} & \mathtt{[}l_1, \ldots, l_k\mathtt{]}
\end{array}
$$

Implementation issues

- Quick and easy-to-implement approach:

  Matching Scheme

  - transformation rules for common situations
  - number of `append`s, pivot positions, ...

    Example: inorder

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

- `quicksort`, `reverse`, `hanoi`, `pre/in/postTree`, ...

```
qs (x:xs) = append (qs l) (x:qs r)
              where (l,r) = split x xs
```

$\Downarrow$

```
qs_acc (x:xs) ys = qs_acc l (x:qs_acc r ys)
                     where (l,r) = split x xs
```

Runtime Figures

- Scheme implemented for and in Curry
- `http://www.dsic.upv.es/users/elp/clop/`

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

Benchmark results (in ms.)

Conclusions

Related work:

- Difference-List Transformation
  (Marriot, Søndergaard '88)
- The concatenate vanishes
  (Wadler '87, Sands '96, Gill '96)
  - different approach (rewrite rules)
  - same effect, maybe different class?
  - implementation likely more inefficient

Future work:

- study larger class of programs
  $\Rightarrow$ maybe implement the full algorithm
- detailed comparison to Wadler
- improvement of the matching scheme