The Concatenate Vanishes

Philip Wadler
University of Glasgow*

December 1987 (Revised, November 1989)

This note presents a trivial transformation that can eliminate many calls of the concatenate (or "append") operator from a program. The general form of the transformation is well known, and one of the examples, transforming the reverse function, is a classic. However, so far as I am aware, this style of transformation has not previously been systematised in the way done here.

The transformation is suitable for incorporation in a compiler, and improves the asymptotic time complexity of some programs from quadratic to linear. There is a syntactic test that determines when the transformation will succeed in eliminating a concatenate operation.

Section 1 describes the transformation. Section 2 presents examples. Section 3 characterises the transformation's benefits. Section 4 describes related work. Section 5 concludes.

1 The transformation

First, some notational preliminaries. We write concatenate as infix $+\!\!+$, list construction (cons) as infix :, and the empty list as $[\,]$. We write $[x, y, z]$ as an abbreviation for $x : (y : (z : [\,]))$.

We will make use of the following laws:

$$\begin{array}{lll} (1) & [\,] +\!\!+ x & = & x \\ (2) & (x : y) +\!\!+ z & = & x : (y +\!\!+ z) \\ (3) & (x +\!\!+ y) +\!\!+ z & = & x +\!\!+ (y +\!\!+ z) \end{array}$$

Laws (1) and (2) provide a recursive definition of concatenate. Law (3) states that concatenate is associative; it may be proved from laws (1) and (2).

We now describe the transformation. The key idea is that whenever an application of a function f may appear as the left argument of a concatenation, then we introduce a new function f', satisfying the property

$$(*) \quad (f\ x_1\ \ldots\ x_n) +\!\!+ y \ = \ f'\ x_1\ \ldots\ x_n\ y$$

*Author's address: Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. Electronic mail: wadler@cs.glasgow.ac.uk.

An earlier version of this note was distributed to the "fp" electronic mailing list, December 1987.

We assume that the name f' can be derived from the name f, and appears nowhere in the original program.

For simplicity, we will assume that a program script is a collection of equations. For each equation in the original program

$$f\ p_1\ \ldots\ p_n \quad = \quad e$$

defining f, we introduce a new equation

$$f'\ p_1\ \ldots\ p_n\ y \quad = \quad e +\!\!+ y$$

defining f'. Here $p_1, \ldots, p_n$ are patterns, e is an expression, and y is a new variable name. The original definition of f is replaced by

$$f\ x_1\ \ldots\ x_n \quad = \quad f'\ x_1\ \ldots\ x_n\ [\,]$$

It should be clear that the new f is equal to the old f, and that f and f' satisfy $(*)$.

The script is then transformed by repeatedly applying laws (1)–(3) and each property $(*)$ as rewrite rules, replacing each instance of a left-hand side by the corresponding right-hand side. Whenever possible, law (3) should be applied before property $(*)$. If we also use the property

$$(f'\ x_1\ \ldots\ x_n\ y) +\!\!+ z \quad = \quad f'\ x_1\ \ldots\ x_n\ (y +\!\!+ z)$$

as a rewrite rule, then the order in which rules are applied is immaterial.

When no more rewrites can be applied, the transformation is complete.

The validity of this transformation is easy to verify, as it fits directly into Burstall and Darlington's [BD77] classic program transformation approach.

2 Examples

2.1 Reverse

A function to reverse a list is defined by the script

$$\begin{array}{lcl} \mathit{reverse}\ [\,] & = & [\,] \\ \mathit{reverse}\ (x : \mathit{xs}) & = & \mathit{reverse}\ \mathit{xs} +\!\!+ [x] \end{array}$$

To apply the transformation, we first form the new definition

$$\begin{array}{lcl} \mathit{reverse}'\ [\,]\ \mathit{ys} & = & [\,] +\!\!+ \mathit{ys} \\ \mathit{reverse}'\ (x : \mathit{xs})\ \mathit{ys} & = & (\mathit{reverse}\ \mathit{xs} +\!\!+ [x]) +\!\!+ \mathit{ys} \end{array}$$

We now rewrite the right-hand sides of each equation according to the rules given above:

$$\begin{array}{lcll} [\,] +\!\!+ \mathit{ys} & = & \mathit{ys} & (1) \\ (\mathit{reverse}\ \mathit{xs} +\!\!+ [x]) +\!\!+ \mathit{ys} & = & \mathit{reverse}\ \mathit{xs} +\!\!+ ([x] +\!\!+ \mathit{ys}) & (3) \\ & = & \mathit{reverse}\ \mathit{xs} +\!\!+ (x : \mathit{ys}) & (2,1) \\ & = & \mathit{reverse}'\ \mathit{xs}\ (x : \mathit{ys}) & (*) \end{array}$$

This yields the final definition,

$$\begin{array}{lcl} \mathit{reverse}\ \mathit{xs} & = & \mathit{reverse}'\ \mathit{xs}\ [\,] \\ \\ \mathit{reverse}'\ [\,]\ \mathit{ys} & = & \mathit{ys} \\ \mathit{reverse}'\ (x : \mathit{xs})\ \mathit{ys} & = & \mathit{reverse}'\ \mathit{xs}\ (x : \mathit{ys}) \end{array}$$

Whereas the original definition of *reverse* is quadratic in the length of the input list, the transformed definition is linear.

2.2 Tree traversal

Binary trees can be represented using the constructors Leaf and Branch, so that

$$\mathit{Branch}\ (\mathit{Branch}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 2))\ (\mathit{Leaf}\ 3)$$

is a typical tree. Traversing a tree returns a left-to-right list of its leaves, so traversing the above tree returns [1,2,3].

A function to traverse a tree is defined by the script

$$\begin{array}{lcl} \mathit{traverse}\ (\mathit{Leaf}\ x) & = & [x] \\ \mathit{traverse}\ (\mathit{Branch}\ \mathit{xt}\ \mathit{yt}) & = & \mathit{traverse}\ \mathit{xt} +\!\!+ \mathit{traverse}\ \mathit{yt} \end{array}$$

Applying the transformation (and omitting the details this time) yields the new script

$$\begin{array}{lcl} \mathit{traverse}\ \mathit{xt} & = & \mathit{traverse}'\ \mathit{xt}\ [\,] \\ \\ \mathit{traverse}'\ (\mathit{Leaf}\ x)\ \mathit{zs} & = & x : \mathit{zs} \\ \mathit{traverse}'\ (\mathit{Branch}\ \mathit{xt}\ \mathit{yt})\ \mathit{zs} & = & \mathit{traverse}'\ \mathit{xt}\ (\mathit{traverse}'\ \mathit{yt}\ \mathit{zs}) \end{array}$$

Whereas the original definition of *traverse* is (in the worst case) quadratic in the size of the input tree, the transformed definition is (in all cases) linear.

2.3 Quicksort

The traditional quicksort algorithm is defined by the script

$$\begin{array}{lcl} \mathit{qsort}\ [\,] & = & [\,] \\ \mathit{qsort}\ (x : \mathit{xs}) & = & \mathit{qsort}\ (\mathit{below}\ x\ \mathit{xs}) +\!\!+ [x] +\!\!+ \mathit{qsort}\ (\mathit{above}\ x\ \mathit{xs}) \end{array}$$

where $(\mathit{below}\ x\ \mathit{xs})$ returns a list of all elements in xs that are less than x, and $(\mathit{above}\ x\ \mathit{xs})$ returns a list of all elements in xs that are greater than or equal to x. Applying the transformation yields the new script

$$\begin{array}{lcl} \mathit{qsort}\ \mathit{xs} & = & \mathit{qsort}'\ \mathit{xs}\ [\,] \\ \\ \mathit{qsort}'\ [\,]\ \mathit{ys} & = & \mathit{ys} \\ \mathit{qsort}'\ (x : \mathit{xs})\ \mathit{ys} & = & \mathit{qsort}'\ (\mathit{below}\ x\ \mathit{xs})\ (x : \mathit{qsort}'\ (\mathit{above}\ x\ \mathit{xs})\ \mathit{ys}) \end{array}$$

where *above* and *below* are as before.

The original and the transformed definitions of *qsort* both have the same asymptotic complexity, namely quadratic in the worst case, and $n \log n$ in the average case, where n is the length of the input list. However, the transformed definition has a slightly better absolute performance. Measurements made with the Orwell [WMR87] interpreter show that the transformed definition requires about 10% less in run time, number of reductions performed, and number of cells allocated. Measurements made with the Miranda[1] [Tur85] interpreter show a similar improvement.

2.4 Flattening a list of lists

Here are some examples of scripts not improved by the transformation. A function to concatenate a list of lists is defined by the script

$$\begin{array}{lcl} \mathit{flatten}\ [\,] & = & [\,] \\ \mathit{flatten}\ (xs : xss) & = & xs +\!\!+ \mathit{flatten}\ xss \end{array}$$

The transformation does not alter this script at all. Furthermore, if in the previous scripts we had always written

$$\mathit{flatten}\ [xs, ys]$$

in place of $xs +\!\!+ ys$, then these scripts would not have been improved either.

A confused programmer might write the following definition:

$$\begin{array}{lcl} \mathit{silly}\ xs\ [\,] & = & xs \\ \mathit{silly}\ xs\ (ys : yss) & = & \mathit{silly}\ (xs +\!\!+ ys)\ yss \end{array}$$

If *yss* is a finite list, then

$$\mathit{silly}\ xs\ yss \quad = \quad xs +\!\!+ \mathit{flatten}\ yss$$

However, *flatten* requires time linear in the size of its input, whereas *silly* requires (in the worst case) quadratic time. Again, this script is not altered by the transformation. So there is at least one script involving concatenate (albeit a silly one) that the transformation does *not* improve from quadratic to linear time.

2.5 Tree traversal, revisited

A different sort of tree can be represented using the constructor *Spine*, so that

$$\mathit{Spine}\ 1\ [\mathit{Spine}\ 2\ [\,], \mathit{Spine}\ 3\ [\,]]$$

is a typical tree. Left-to-right traversal of this tree yields [1,2,3].

A function to traverse such trees can be defined by

$$\mathit{traverseTree}\ (\mathit{Spine}\ x\ xts) \quad = \quad x : \mathit{flatten}\ (\mathit{map}\ \mathit{traverseTree}\ xts)$$

where *map* applies a function to each element of a list. Applying the transformation to this script has no effect, so we might consider this another failure of the transformation technique.

[1] Miranda is a trademark of Research Sofware Limited.

However, if we first expand the above definition into the form

$$\begin{array}{lcl} \mathit{traverseTree}\ (\mathit{Spine}\ x\ \mathit{xts}) & = & x : \mathit{traverseForest}\ \mathit{xts} \\ \\ \mathit{traverseForest}\ [\,] & = & [\,] \\ \mathit{traverseForest}\ (\mathit{xt} : \mathit{xts}) & = & \mathit{traverseTree}\ \mathit{xt} +\!\!+ \mathit{traverseForest}\ \mathit{xts} \end{array}$$

then the resulting script will be transformed in a way similar to the previous traversal example, in this case yielding two new functions, $\mathit{traverseTree}'$ and $\mathit{traverseForest}'$. It may be possible for a compiler to perform automatically an expansion like the one given here, but we leave exploration of that issue to a future paper.

3 Characterising the transformation

It is clear that this tranformation could easily be incorporated into a compiler for a functional language, and the examples suggest this may yield a significant benefit. However, any transformation introduced into a compiler will be useless if it affects the behaviour of the transformed programs in a way that is difficult to predict. Of course, one could always determine the behaviour by examining the transformed script, but it is better to be able to analyse the program in terms of the source script. This section presents a method of doing so.

To do this, we need to classify each list-valued expression and function in a script as either *creative* or *plagiarizing*. An expression is creative if it allocates each cons cell in its result, as in $[x, y, z]$, and plagiarizing if some cons cells in the result are shared with a free-variable of the expression, as in $x : y : zs$. Similarly, a function definition is creative if it allocates each cons cell in the result, and plagiarizing if some cons cells are shared with an argument of the function. As a simple example, given the script

$$\begin{array}{lcl} \mathit{copy}\ [\,] & = & [\,] \\ \mathit{copy}\ (x : \mathit{xs}) & = & x : \mathit{copy}\ \mathit{xs} \\ \\ \mathit{concat}\ [\,]\ \mathit{ys} & = & \mathit{ys} \\ \mathit{concat}\ (x : \mathit{xs})\ \mathit{ys} & = & x : \mathit{concat}\ \mathit{xs}\ \mathit{ys} \end{array}$$

then *copy* is creative, whereas *concat* is plagiarizing.

There is a simple syntactic test for creativity. Let F be a set of functions assumed to have creative definitions. An expression is *creative relative to* F if it is $[\,]$ or has one of the forms

$e_1 : e_2$	and e_2 is creative relative to F
$e_1 +\!\!+ e_2$	and e_1 and e_2 are creative relative to F
$f\ e_1\ \ldots\ e_n$	and f is in F

There are no requirements on $e_1, \ldots, e_n$ in the last case. An expression consisting only of a variable is not creative.

Given a script S_0, let F_0 be the largest set such that for each f in F_0, all the right-hand sides of f in S_0 are creative relative to F_0. Then the creative function definitions in S_0 are just those in F_0, and the creative expressions in S_0 are just those that are creative

relative to F_0. By this test, the definitions given above for *reverse*, *traverse*, and *qsort* are creative, while the definition of *flatten* is not.

We can now characterize the benefits of the transformation for a given script. In each expression of the form $(e_1 +\!\!+ e_2)$, if e_1 is creative then the occurence of $+\!\!+$ will be transformed out of the script. Further, when computing asymptotic time complexity, in each expression of the form $(e_1 +\!\!+ e_2)$, if e_1 is creative then the time to perform the $+\!\!+$ operation may be taken as constant rather than proportional to the length of e_1. Using this model, we can predict that *reverse* and *traverse*, as given above, will improve from quadratic to linear, whereas *qsort* will retain the same asymptotic time complexity.

4 Related work

Sleep and Holmström [SH82] have described an ingenious implementation of concatenate. Their run-time implementation is more flexible than the compile-time transformation described here, but it incurs higher overhead. For example, under their method the untransformed versions of *reverse* and *traverse* will run in linear (asymptotic) time, but with higher (absolute) time overhead than the equivalent transformed versions. On the other hand, the transformation described here leaves *silly* unchanged, and hence requiring quadratic time, while under their method it runs in linear time. There is no harm in using both methods: in this case, concatenate can always be taken as requiring constant asymptotic time, while the model in Section 3 now describes when run-time overhead will be eliminated.

The transformation given here has much the same effect as the one described in [Hug86]. The main difference is that the transformation here is described in such a way that it could be added to a compiler, and a precise characterisation of the effect on efficiency has been given.

It may be possible to combine the transformation described here with the "deforestation" transformation [Wad88]. Doing so may eliminate some of the shortcomings described in Sections 2.4 and 2.5.

5 Conclusions

A simple algorithm has been presented that removes many instances of concatenate from a program. The algorithm improves some familiar algorithms, such as reversing a list or traversing a binary tree, from quadratic to linear in terms of asymptotic time complexity. It improves some others, like quicksort, by a constant factor. It is possible to characterise those instances where the algorithm will succeed in eliminating concatenate, and to characterise the resulting improvement (if any) in asymptotic time complexity.

Functions that convert data structures into strings often look similar to the tree traversal functions transformed here, and thus are likely to yield notable improvements. John O'Donnell tells an anecdote about a pretty-printer he once wrote, that required several hours to run. When he applied (by hand) a transformation similar to the one described here, the run time was reduced to less than ten minutes.

The transformation presented here is astonishingly simple. The first reaction on seeing it is "that's obvious". However, I've never before heard it suggested that applications of concatenate can be transformed away automatically, and I know of no compiler that incorporates this transformation.

Would it be too much to suggest that the transformation is obvious only in retrospect? Are there other obvious transformations waiting to be discovered?

Acknowledgements

I am grateful to John O'Donnell for the above anecdote, and to John Launchbury for suggesting the quicksort example and helping to measure the improvement in efficiency. I am particularly grateful to David Watt, who asked the critical question I had never considered in a decade of work on program transformation: Is there an algorithm that can make concatenate vanish?

References

[BD77] R. M. Burstall and J. Darlington, A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Hug86] J. Hughes, A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, **22**:141–144, March 1986.

[SH82] M. R. Sleep and S. Holmström, A short note concerning lazy reduction rules of append. *Software Practice and Experience*, 12(11):1082–4, November 1982.

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In G. Kahn, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.

[Wad88] P. L. Wadler, Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *European Symposium On Programming*, Nancy, France, March 1988. LNCS 300, Springer-Verlag, 1988. Revised version to appear in *Theoretical Computer Science*.

[WMR87] P. L. Wadler, Q. Miller, and M. Raskovsky, An introduction to Orwell. Programming Research Group, Oxford University, 1987.