J. Functional Programming 3 (2): 247–250, April 1993

Terminating comprehensions

CHRIS READE
Computer Science Department, Brunel University, Uxbridge, Middx. UB8 3PH, UK
and
S.E.D. Informatics Department, Rutherford Appleton Laboratory, Chilton, Didcot, Oxon OX11 0QX, UK

Abstract

List terminators are discussed as a new form of qualifier in list comprehensions for early termination of a list. The semantics of list terminators is expressed in terms of an optimal translation of list comprehensions (cf. Wadler, 1987) because it makes direct use of a continuation list.

List comprehensions, as used in functional languages such as Haskell (Hudak *et al.*, 1992), are usually of the form $[u \mid q_1, \ldots, q_n]$ where u is an expression and q_i are qualifiers. Each qualifier is either a generator of the form $x \leftarrow t$, where x is a variable and t an expression, or a filter of the form b (a boolean expression). For example

$$[(x, y) \mid x \leftarrow [1 \,..\, m], y \leftarrow [1 \,..\, n], x \neq y]$$

lists the co-ordinates of an m by n matrix excluding the diagonal.

A second form of boolean test as a qualifier is proposed here, whose purpose is to terminate the list when a condition fails rather than to filter out elements when the condition fails. The keyword **while** will be used to distinguish these termination tests. For example, we might write

$$[\mathit{Appendchan}\ \mathit{stdout}\ (\mathit{display}\ (f\ n)) \mid s \leftarrow \mathit{lines}\ \mathit{userInput}, \ \textbf{while}\ s \neq [\,], n \leftarrow \mathit{findInts}\ s]$$

as part of a program to display $f\,n$ for integers n entered by the user, terminating when an empty line is found. As another example, consider the definition of a function to convert positive integers to strings

$$\mathit{stringofPosint}\ n = \mathit{reverse}[\mathit{charofDigit}\ (i \bmod 10) \mid i \leftarrow \mathit{iterate}\ (\mathit{div}\ 10)\ n, \textbf{while}\ i \neq 0]$$

where $\mathit{iterate}\ (\mathit{div}\ 10)\ n$ produces $[n, n \mathit{div} 10, (n \mathit{div} 10) \mathit{div} 10, \ldots]$ and the terminator is used to cut this list off before the first occurrence of 0.

Sometimes, such a termination can be achieved using a function *takeWhile* outside the list comprehension. This function can be regarded as a special case, where

$$\mathit{takeWhile}\ p\ xs = [x \mid x \leftarrow xs, \textbf{while}\ p\ x].$$

https://doi.org/10.1017/S095679680000071X Published online by Cambridge University Press

However, if the terminator is embedded in the middle of a comprehension, it may not be so simple to re-express it with *takeWhile*. As a simple concrete example, consider the expression

$$[z \mid x \leftarrow [1\,..], y \leftarrow [1\,..\,x], \textbf{while } y < 4, z \leftarrow [1\,..\,y]]$$

which would produce

$$[1,1,1,2,1,1,2,1,2,3,1,1,2,1,2,3].$$

In order to convert this using an application of *takeWhile*, we would have to split the comprehension into two phases (generating the x and y values before the test using *takeWhile*, and the subsequent generation of the z values), e.g.

$$[z \mid y \leftarrow takeWhile\ (<4)\ [y \mid x \leftarrow [1\,..], y \leftarrow [1\,..\,x]], z \leftarrow [1\,..\,y]]$$

which loses some clarity.

We show below that the meaning of terminators relies on a continuation style semantics rather than a direct semantics. This is analogous to constructs such as *callcc* and *goto* which have their uses, but also complicate the semantics. Rather than include such a construct in a programming language implementation, it is probably better for programmers just to be aware of the translation. This would allow terminators to be used while developing an algorithm, and translated out by hand (assuming the use is infrequent). Indeed, for functional languages which do not support comprehensions (such as SML) use of comprehensions along with hand translations is still recommended for algorithm development (Reade, 1989).

The semantics for comprehensions are usually expressed with the following equations (Wadler, 1990)

$$[u \mid] = [u] \tag{1}$$

$$[u \mid ps, qs] = concat\ [[u \mid qs] \mid ps] \tag{2}$$

$$[u \mid x \leftarrow t] = map\ (\lambda x\,.\,u)\ t \tag{3}$$

where $map :: (a \to b) \to [a] \to [b]$ applies a function to each item in a list and $concat :: [[a]] \to [a]$ joins a list of lists into a single list (see, e.g., Hudak *et al.*, 1992 for definitions).

In equation (2), ps and qs stands for sequences of qualifiers and ps, qs is the concatenation of these sequences. (The choice of where to break a sequence into ps and qs does not affect the semantics.)

Equations (1)–(3) can be used as left to right rewrite rules to translate list comprehensions. Although rule 2 is applicable when ps or qs is empty, these cases should be avoided in rewriting to prevent redundant or non-terminating translations. To deal with filters, we just add

$$[u \mid b] = \textbf{if } b \textbf{ then } [u] \textbf{ else } [\,]. \tag{4}$$

In Wadler (1987) more primitive one-step reduction rules are given for comprehensions along with a derivation of an optimal translation using a list continuation $(+\!\!+c)$ – a list to be appended at the end which will initially be []. We re-express the optimal translation with the following equations

$$[u \mid x \leftarrow t, qs] +\!\!+ c = foldr\ (\lambda x\,.\,\lambda new\,.\,[u \mid qs] +\!\!+ new)\ c\ t \tag{5}$$

https://doi.org/10.1017/S095679680000071X Published online by Cambridge University Press

$$[u\,|\,]{+\!\!+}c = u{:}c \tag{6}$$

$$[u\,|\,b, qs]{+\!\!+}c = \textbf{if } b \textbf{ then } [u\,|\,qs]{+\!\!+}c \textbf{ else } c \tag{7}$$

where (++) is list append, (:) is list cons, and *foldr* is a standard list operation for combining elements of a list using a (two argument) function and a default value for the empty list.

In equation (5), a variable *new* is introduced which should be distinct from any free variables in the comprehension being translated to avoid a clash. Note that these new equations are more deterministic in that they no longer allow a choice in decomposing a sequence of qualifiers. Equations (5)–(7) can be obtained from equations (1)–(4) by doing some simple calculations. The key properties needed are

$$foldr\, g\, a\, (map\, f\, xs) = foldr\, (g \circ f)\, a\, xs$$

$$foldr\, (+\!\!+)\, c\, xs = (concat\, xs){+\!\!+}c.$$

The new terminator construct can be expressed easily as an addition to rules (5)–(7) (but not directly using (1)–(3))

$$[u\,|\,\textbf{while } b, qs]{+\!\!+}c = \textbf{if } b \textbf{ then } [u\,|\,qs]{+\!\!+}c \textbf{ else } [\,]. \tag{8}$$

This should be compared to equation (7) for filters. The only difference is in the **else** clause, which produces the continuation list for filters and just the empty list for terminators.

As an example, the comprehension

$$[z\,|\,x \leftarrow [1\,..\,], y \leftarrow [1\,..\,x], \textbf{while } y < 4, z \leftarrow [1\,..\,y]]$$

translates as

$$\begin{aligned} foldr\ (\lambda x.\lambda new.foldr\ \ (\lambda y.\lambda new.\ \ &\textbf{if } y < 4 \\ &\textbf{then } foldr\ \ (\lambda z.\lambda new.\ z{:}new \\ &\qquad)\, new\, [1\,..\,y] \\ &\textbf{else } [\,] \\ )\, new\, [1\,..\,x] & \\ )\, [\,]\, [1\,..\,]. & \end{aligned}$$

Finally, a note of caution. The introduction of terminators *forces* a switch from the semantics given by rules (1)–(4) to the semantics given by rules (5)–(8) because rule (2) is no longer valid. For example,

$$[x\,|\,x \leftarrow [1\,..\,], \textbf{while } x < 2] = [1]$$

but

$$[[x\,|\,\textbf{while } x < 2]\,|\,x \leftarrow [1\,..\,]] = 1{:}\bot.$$

Acknowledgements

I thank Dave Martland for pointing out the need for list terminators, Brian Ritchie for useful discussions, and Phil Wadler for advice on improving the presentation.

References

Hudak, P., Peyton Jones, S., Wadler, P. eds. *et al.*, 1992. The Haskell Report. *ACM SIGPLAN Notices*, **27**(5): May.

Reade, C. M. P. 1989. *Elements of Functional Programming*. Addison Wesley.

Wadler, P. 1987. List comprehensions. In S. Peyton Jones ed., *The Implementation of Functional Programming Languages*. Prentice Hall.

Wadler, P. 1990. Comprehending monads. In *Proceedings ACM Conference on Lisp and Functional Programming*, Nice, France.