Stream Fusion

From Lists to Streams to Nothing at All

Duncan Coutts[1] Roman Leshchinskiy[2] Don Stewart[2]

[1] Programming Tools Group
Oxford University Computing Laboratory
duncan.coutts@comlab.ox.ac.uk

[2] Computer Science & Engineering
University of New South Wales
{rl,dons}@cse.unsw.edu.au

Abstract

This paper presents an automatic deforestation system, *stream fusion*, based on equational transformations, that fuses a wider range of functions than existing short-cut fusion systems. In particular, stream fusion is able to fuse zips, left folds and functions over nested lists, including list comprehensions. A distinguishing feature of the framework is its simplicity: by transforming list functions to expose their structure, intermediate values are eliminated by general purpose compiler optimisations.

We have reimplemented the Haskell standard List library on top of our framework, providing stream fusion for Haskell lists. By allowing a wider range of functions to fuse, we see an increase in the number of occurrences of fusion in typical Haskell programs. We present benchmarks documenting time and space improvements.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.4 [*Programming Languages*]: Optimization

General Terms Languages, Algorithms

Keywords Deforestation, program optimisation, program transformation, program fusion, functional programming

1. Introduction

Lists are the primary data structure of functional programming. In lazy languages, such as Haskell, lists also serve in place of traditional control structures. It has long been recognised that composing list functions to build programs in this style has advantages for clarity and modularity, but that it incurs some runtime penalty, as functions allocate intermediate values to communicate results. Fusion (or deforestation) attempts to remove the overhead of programming in this style by combining adjacent transformations on structures to eliminate intermediate values.

Consider this simple function which uses a number of intermediate lists:

$$f :: Int \to Int$$
$$f\ n\ =\ sum\,[\,k * m \mid k\ \leftarrow\ [1..n],\ m\ \leftarrow\ [1..k]\,]$$

No previously implemented short-cut fusion system eliminates all the lists in this example. The fusion system presented in this paper does. With this system, the Glasgow Haskell Compiler (The GHC Team 2007) applies all the fusion transformations and is able to generate an efficient "worker" function f' that uses only unboxed integers (*Int#*) and runs in constant space:

```
f' :: Int# → Int#
f' n =
  let go :: Int# → Int# → Int#
      go z k =
        case k > n of
          False → case 1 > k of
                    False → to (z + k) k (k + 1) 2
                    True  → go z (k + 1)
          True  → z
      to :: Int# → Int# → Int# → Int# → Int#
      to z k k' m =
        case m > k of
          False → to (z + (k * m)) k k' (m + 1)
          True  → go z k'
  in go 0 1
```

Stream fusion takes a simple three step approach:

1. Convert recursive structures into non-recursive *co-structures*;
2. Eliminate superfluous conversions between structures and co-structures;
3. Finally, use general optimisations to fuse the co-structure code.

By transforming pipelines of recursive list functions into non-recursive ones, code becomes easier to optimise, producing better results. The ability to fuse all common list functions allows the programmer to write in an elegant declarative style, and still produce excellent low level code. We can finally write the code we *want* to be able to write without sacrificing performance!

1.1 Short-cut fusion

The example program is a typical high-level composition of list producers, transformers and consumers. However, extensive optimisations are required to transform programs written in this style into efficient low-level code. In particular, naive compilation will produce a number of intermediate data structures, resulting in poor performance. We would like to have the compiler remove these intermediate structures automatically. This problem, deforestation (Wadler 1990), has been studied extensively (Meijer et al. 1991; Gill et al. 1993; Takano and Meijer 1995; Gill 1996; Hu et al. 1996; Chitil 1999; Johann 2001; Svenningsson 2002; Gibbons 2004). To illustrate how our approach builds on previous work on short-cut fusion, we review the main approaches.