ARTICLE IN PRESS
SCICO:2037

Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Multiple intermediate structure deforestation by shortcut fusion

Alberto Pardo [a,*], João Paulo Fernandes [b,c], João Saraiva [c]

[a] Instituto de Computación, Universidad de la República, Uruguay
[b] LISP/ ((rel)ease) Reliable and Secure Computation Group, Universidade da Beira Interior, Portugal
[c] HASLab/INESC TEC, Universidade do Minho, Portugal

ARTICLE INFO

Article history:
Received 18 April 2014
Received in revised form 19 July 2016
Accepted 19 July 2016
Available online xxxx

Keywords:
Shortcut fusion
Circular programming
Deforestation
Functional programming

ABSTRACT

Shortcut fusion is a well-known optimization technique for functional programs. Its aim is to transform multi-pass algorithms into single pass ones, achieving deforestation of the intermediate structures that multi-pass algorithms need to construct. Shortcut fusion has already been extended in several ways. It can be applied to monadic programs, maintaining the global effects, and also to obtain circular and higher-order programs. The techniques proposed so far, however, only consider programs defined as the composition of a single producer with a single consumer. In this paper, we analyse shortcut fusion laws to deal with programs consisting of an arbitrary number of function compositions.

1. Introduction

Shortcut fusion [1] is an important optimization technique for functional programs. It was proposed as a technique for the elimination of intermediate data structures generated in function compositions $fc \circ fp$, where a producer $fp :: a \rightarrow t$ builds a data structure t, which is then traversed by a consumer $fc :: t \rightarrow b$ to produce a result. When some conditions are satisfied, we may transform these programs into equivalent ones that do not construct the intermediate structure of type t.

Extended forms of shortcut fusion have also been proposed to cope with cases of function compositions in which the producer and the consumer communicate through some additional context information, besides the intermediate structure itself. These extensions transform compositions $fc \circ fp$, where $fp :: a \rightarrow (t, z)$ and $fc :: (t, z) \rightarrow b$, into circular [2,3] and higher-order [3,4] equivalent programs, increasing the applicability scope of shortcut fusion. Nevertheless, they consider programs consisting of the composition between two functions only. As a consequence, it is not possible to (straightforwardly) apply such techniques to programs that rely on multiple traversal strategies, like compilers and advanced pretty-printing algorithms [5].

The main contribution of this paper is to present generalized forms of shortcut fusion which apply to an arbitrary number of function compositions of the form $f_n \circ \cdots \circ f_0$, for $n >= 2$. We establish sufficient conditions on each f_i that guarantee that consecutive fusion steps are applicable when following both a left-to-right and a right-to-left strategy. By means of what we call *chain laws*, we show how to obtain the intermediate fused definitions in such a way that further fusion steps apply. The formulation of the chain laws is the result of combining two fusion approaches: that of shortcut fusion and the one used in the formulation of fusion laws known as *acid rain* [6]. The fusion laws we present are surely not

* Corresponding author.
E-mail addresses: pardo@fing.edu.uy (A. Pardo), jpf@di.ubi.pt (J.P. Fernandes), jas@di.uminho.pt (J. Saraiva).

http://dx.doi.org/10.1016/j.scico.2016.07.004
0167-6423/

Please cite this article in press as: A. Pardo et al., Multiple intermediate structure deforestation by shortcut fusion, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.07.004

surprising, but, on the other hand, have the merit of dealing with nonstandard cases. Our fusion method, characterized by requiring certain conditions on the functions involved in the compositions, differs from that employed by *warm fusion* [7].

In this work our approach is essentially theoretical. We study the formalities associated with the formulation of the new laws, but we do not include performance results and benchmarks related with the application of the laws in practice.

We analyse two forms of multi-traversal programs: a) the standard case where only a data structure is passed between producer and consumer, and b) programs where in each composition, besides a data structure, some additional information is passed. Case (b) is particularly interesting because it is the case where it is possible to derive circular and higher-order programs by the application of fusion.

The fact that we are able of deriving circular programs from multiple function compositions also has strong connections with well-established research on Attribute Grammars (AG) [8]. Indeed, as Johnson [9] and Kuiper and Swierstra [10] originally showed, attribute grammars are naturally implemented in a lazy setting as circular programs. In the AG community, however, a program that relies on multiple function compositions (so, on multiple intermediate data structures) is usually derived from an AG (i.e., from a circular program) using advanced attribute scheduling algorithms [11,12] whose correctness is hard to establish in a formal way. By studying and proving the correctness of the opposite transformation, that is, how a circular program (i.e., an AG) can be derived from a program based on multiple function compositions, we hope to shed even more light into the relationship between attribute grammars/circular programs and their non-lazy implementations/equivalents.

Throughout the paper we will use Haskell notation, assuming a semantics in terms of pointed cpos (complete partial orders), but without the presence of the *seq* function [13].

The paper is structured as follows. In order to make the contents more accessible to a wider audience, we have decided to divide the paper into two parts. The first part, composed by Sections 2–4, develops the whole concepts and laws, but tailored to the specific case of the list type. In this part laws are presented without proofs. Section 2 reviews shortcut fusion, while Section 3 does so with the derivation of circular and higher-order programs. In Section 4 we discuss laws that pose the sufficient conditions for fusing multi-traversal programs over lists. The second part is formed by Sections 5–6 and is where the generic formulation of the concepts and laws developed is presented. By generic we mean valid for a wide class of datatypes. In Section 5 we present the theoretical concepts necessary for building the generic definitions. Section 6 discusses the laws necessary for fusing multi-traversal programs, but now over arbitrary data structures. A proof is presented for each of these laws. Finally, Section 7 concludes the paper.

2. Shortcut fusion on lists

Shortcut fusion [1] is a program transformation technique for the elimination of intermediate data structures generated in function compositions. In this section we focus on the elimination of lists as intermediate data structures. The formulation for arbitrary data structures is described later in Section 5.

For its application, shortcut fusion requires the consumer to process all the elements of the intermediate data structure in a uniform way. This condition is established by requiring that the consumer is expressible as a *fold* [14], a program scheme that captures function definitions by structural recursion. For example, for lists, *fold* is defined as:

$$\begin{array}{ll} fold & :: (b, a \to b \to b) \to [\,a\,] \to b \\ fold\ (nil, cons)\ [\,] & = nil \\ fold\ (nil, cons)\ (a : as) & = cons\ a\ (fold\ (nil, cons)\ as) \end{array}$$

This function corresponds to the well-known *foldr* function [14]. It traverses the list and replaces [ ] by the constant *nil* and the occurrences of (:) by function *cons*. The pair $(nil, cons)$ is called an algebra. The formal definition of an algebra is given in Section 5.

For example, the function *filter*, which selects the elements of a list that satisfy a given predicate:

$$\begin{array}{ll} filter & :: (a \to Bool) \to [\,a\,] \to [\,a\,] \\ filter\ p\ [\,] & = [\,] \\ filter\ p\ (a : as) & = \textbf{if}\ p\ a\ \textbf{then}\ a : filter\ p\ as\ \textbf{else}\ filter\ p\ as \end{array}$$

can be written in terms of *fold* as follows:

$$\begin{array}{l} filter\ p = fold\ (fnil, fcons) \\ \quad \textbf{where}\ fnil = [\,] \\ \qquad\quad fcons\ a\ r = \textbf{if}\ p\ a\ \textbf{then}\ a : r\ \textbf{else}\ r \end{array}$$

The producer, on the other hand, must be a function such that the computation that builds the output data structure consists of the repeated application of the data type constructors. To meet this condition the producer is required to be expressible in terms of a function, called *build* [1], which carries a "template" (a *producer skeleton* as called by Chitil [15]) that abstracts the occurrences of the constructors of the intermediate data type. To guarantee correctness, the template should be a polymorphic function. In the case of lists, *build* is defined as follows:

$$\begin{array}{l} build\ :: (\forall\, b\ .\ (b, a \to b \to b) \to c \to b) \to c \to [\,a\,] \\ build\ g = g\ in \end{array}$$