The Journal of Logic and Algebraic Programming 83 (2014) 53–63

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

www.elsevier.com/locate/jlap

Conversion to tail recursion in term rewriting ☆

Naoki Nishida [a], Germán Vidal [b,*]

[a] Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
[b] MiST, DSIC, Universitat Politècnica de València, Camino de Vera, s/n, 46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 30 October 2012
Received in revised form 8 April 2013
Accepted 3 July 2013
Available online 9 July 2013

Keywords:
Term rewriting
Program transformation
Tail recursion

ABSTRACT

Tail recursive functions are a special kind of recursive functions where the last action in their body is the recursive call. Tail recursion is important for a number of reasons (e.g., they are usually more efficient). In this article, we introduce an automatic transformation of first-order functions into tail recursive form. Functions are defined using a (first-order) term rewrite system. We prove the correctness of the transformation for constructor-based reduction over constructor systems (i.e., typical first-order functional programs).

1. Introduction

Tail recursive functions are recursive functions that call themselves (or other mutually recursive functions) as a final action in their recursive definitions. Tail recursion is specially important because it often makes functions more efficient. From a compiler perspective, tail recursive functions are considered as iterative constructs since the allocated memory in the stack does not grow with the recursive calls (moreover, some functions may become much more efficient in tail recursive form thanks to the use of accumulators). Furthermore, a transformation to tail recursive form may also be useful to define program analyses and transformations that deal with programs in a given *canonical* form (e.g., where all functions are assumed to be tail recursive). This is the case, for instance, of the function inversion technique of [1] that is defined for tail recursive functions.

Let us illustrate the proposed transformation with an example. Consider the following function to concatenate two lists:

$$\mathsf{app}(\mathsf{nil}, y) \to y$$
$$\mathsf{app}(x : xs, y) \to x : \mathsf{app}(xs, y)$$

where nil denotes an empty list and $x : xs$ a list with head x and tail xs. The function app can be transformed into tail recursive form, e.g., as follows:

$$\mathsf{app}(x, y) \to \mathsf{app}'(x, y, \mathsf{id})$$
$$\mathsf{app}'(\mathsf{nil}, y, k) \to \mathsf{eval}(k, y)$$
$$\mathsf{app}'(x : xs, y, k) \to \mathsf{app}'\big(xs, y, \mathsf{cont}(k, x)\big)$$

☆ This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación (Secretaría de Estado de Investigación)* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant PROMETEO/2011/052, and by *MEXT KAKENHI* #21700011.

* Corresponding author.
E-mail addresses: nishida@is.nagoya-u.ac.jp (N. Nishida), gvidal@dsic.upv.es (G. Vidal).

1567-8326/$ – see front matter
http://dx.doi.org/10.1016/j.jlap.2013.07.001

$$\mathsf{eval}(\mathsf{id}, y) \to y$$
$$\mathsf{eval}\big(\mathsf{cont}(k, x), y\big) \to \mathsf{eval}(k, x : y)$$

Intuitively speaking, the essence of the transformation consists in introducing two new *constructor* symbols, a 0-ary constructor id (to stop the construction of *contexts*) and a binary constructor cont (to reconstruct *contexts*). Therefore, a derivation like

$$\mathsf{app}(1 : \mathsf{nil}, 2 : 3 : \mathsf{nil}) \to 1 : \mathsf{app}(\mathsf{nil}, 2 : 3 : \mathsf{nil}) \to 1 : 2 : 3 : \mathsf{nil}$$

becomes

$$\begin{aligned}\mathsf{app}(1 : \mathsf{nil}, 2 : 3 : \mathsf{nil}) &\to \mathsf{app}'(1 : \mathsf{nil}, 2 : 3 : \mathsf{nil}, \mathsf{id})\\ &\to \mathsf{app}'\big(\mathsf{nil}, 2 : 3 : \mathsf{nil}, \mathsf{cont}(\mathsf{id}, 1)\big) \to \mathsf{eval}\big(\mathsf{cont}(\mathsf{id}, 1), 2 : 3 : \mathsf{nil}\big)\\ &\to \mathsf{eval}(\mathsf{id}, 1 : 2 : 3 : \mathsf{nil}) \to 1 : 2 : 3 : \mathsf{nil}\end{aligned}$$

in the transformed program. Observe that, in contrast to the approach of [2], our tail recursive functions are not more efficient than the original ones (actually, they perform some more steps—by a constant factor—due to the introduction of the auxiliary function eval).[1] This is similar to the introduction of *continuations* [3,4] in higher-order λ-calculus, i.e., lambda expressions that encode the future course of a computation. For instance, the example above would be transformed using continuations as follows:

$$\mathsf{app}\, xy \to \mathsf{app}_{\mathsf{c}}\, xy(\lambda w.w)$$
$$\mathsf{app}_{\mathsf{c}}\, \mathsf{nil}\, yk \to ky$$
$$\mathsf{app}_{\mathsf{c}}\, (x : xs)\, yk \to \mathsf{app}_{\mathsf{c}}\, xsy\big(\lambda w.k(x : w)\big)$$

As in our approach, some more steps are required in order to reduce the continuations:

$$\begin{aligned}\mathsf{app}(1 : \mathsf{nil})(2 : 3 : \mathsf{nil}) &\to \mathsf{app}_{\mathsf{c}}\, (1 : \mathsf{nil})(2 : 3 : \mathsf{nil})(\lambda w.w)\\ &\to \mathsf{app}_{\mathsf{c}}\, \mathsf{nil}(2 : 3 : \mathsf{nil})\big(\lambda w'.(\lambda w.w)\big(1 : w'\big)\big)\\ &\to \big(\lambda w'.(\lambda w.w)\big(1 : w'\big)\big)(2 : 3 : \mathsf{nil})\\ &\to (\lambda w.w)(1 : 2 : 3 : \mathsf{nil}) \to 1 : 2 : 3 : \mathsf{nil}\end{aligned}$$

In this article, we introduce an automatic transformation of first-order functions to tail recursive form. In our setting, functions are defined using a constructor term rewrite system, i.e., a typical (first-order) functional program. Moreover, in order to preserve the semantics through the transformation, we only consider a particular form of innermost reductions (i.e., call by value) that is known as constructor-based reduction.

As mentioned before, in the context of λ-calculus, a similar goal can be achieved by introducing continuations [3,4]. However, this implies moving to a higher-order setting and we aim at defining a first-order transformation. Hence we share the aim with Wand's seminal paper [5], though he focused on improving the complexity of functions by introducing accumulators (*a data structure representing a continuation function*, in Wand's words). However, the examples presented by Wand required some *eureka* steps and, therefore, no automatic technique is introduced. A similar approach is also presented by Field and Harrison [2], where the function accumulators are derived (manually) from functions with continuations. Although the introduction of accumulators is out of the scope of this paper, an example illustrating such a transformation in our context can be found in Section 4.

The paper is organized as follows. In Section 2 we briefly review some notions and notations from term rewriting. Section 3 presents our transformation for converting functions to tail recursive form and proves its correctness. Finally, Section 4 concludes and points out a challenging direction for future research.

2. Preliminaries

In this section, we recall some basic notions and notations of term rewriting [6,7].

Throughout this paper, we use $\mathcal{V}$ as a countably infinite set of *variables*. Let $\mathcal{F}$ be a *signature*, i.e., a finite set of *function symbols* with a fixed arity denoted by $\mathrm{ar}(f)$ for a function symbol f. We often write f/n to denote a function symbol f with arity $\mathrm{ar}(f) = n$. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and the set of variables appearing in terms $t_1, \ldots, t_n$ is denoted by $\mathcal{V}ar(t_1, \ldots, t_n)$. The notation $C[t_1, \ldots, t_n]_{p_1,\ldots,p_n}$ represents the term obtained by replacing each *hole* $\Box$ at *position* p_i of an n-hole *context* $C[\,]$ with term t_i for all $1 \leqslant i \leqslant n$. We may omit the subscripts $p_1, \ldots, p_n$ when they

[1] Nevertheless, it may run faster in general thanks to the use of tail recursion.