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

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).

Keywords: term rewriting, program transformation, tail recursion

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

[☆]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.
This paper is published in "*Naoki Nishida, Germán Vidal: Conversion to tail recursion in term rewriting. Journal of Logic and Algebraic Programming 83(1): 53-63 (2014).*" DOI: `http://dx.doi.org/10.1016/j.jlap.2013.07.001`. © Elsevier

Email addresses: `nishida@is.nagoya-u.ac.jp` (Naoki Nishida), `gvidal@dsic.upv.es` (Germán Vidal) 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:

$$\begin{array}{rcl} \mathsf{app}(\mathsf{nil}, y) & \to & y \\ \mathsf{app}(x : xs, y) & \to & x : \mathsf{app}(xs, y) \end{array}$$

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:

$$\begin{array}{rcl} \mathsf{app}(x, y) & \to & \mathsf{app}'(x, y, \mathsf{id}) \\ \mathsf{app}'(\mathsf{nil}, y, k) & \to & \mathsf{eval}(k, y) \\ \mathsf{app}'(x : xs, y, k) & \to & \mathsf{app}'(xs, y, \mathsf{cont}(k, x)) \\ \mathsf{eval}(\mathsf{id}, y) & \to & y \\ \mathsf{eval}(\mathsf{cont}(k, x), y) & \to & \mathsf{eval}(k, x : y) \end{array}$$

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

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

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

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

that encode the future course of a computation. For instance, the example above would be transformed using continuations as follows:

$$\begin{array}{rcl} \mathsf{app}\ x\ y & \to & \mathsf{app_c}\ x\ y\ (\lambda w.\ w) \\ \mathsf{app_c}\ \mathsf{nil}\ y\ k & \to & k\ y \\ \mathsf{app_c}\ (x : xs)\ y\ k & \to & \mathsf{app_c}\ xs\ y\ (\lambda w.\ k\ (x : w)) \end{array}$$

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

$$\begin{array}{l} \mathsf{app}\ (\mathsf{1} : \mathsf{nil})\ (\mathsf{2} : \mathsf{3} : \mathsf{nil}) \to \mathsf{app_c}\ (\mathsf{1} : \mathsf{nil})\ (\mathsf{2} : \mathsf{3} : \mathsf{nil})\ (\lambda w.\ w) \\ \to \mathsf{app_c}\ \mathsf{nil}\ (\mathsf{2} : \mathsf{3} : \mathsf{nil})\ (\lambda w'.\ (\lambda w.\ w)\ (\mathsf{1} : w')) \\ \to (\lambda w'.\ (\lambda w.\ w)\ (\mathsf{1} : w'))\ (\mathsf{2} : \mathsf{3} : \mathsf{nil}) \\ \to (\lambda w.\ w)\ (\mathsf{1} : \mathsf{2} : \mathsf{3} : \mathsf{nil}) \to \mathsf{1} : \mathsf{2} : \mathsf{3} : \mathsf{nil} \end{array}$$

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 $\mathsf{ar}(f)$ for a function symbol f. We often write f/n to denote a function symbol f with arity $\mathsf{ar}(f) = n$. 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* $\square$ at *position* p_i of an n-hole *context* $C[\,]$ with term t_i for all $1 \leq i \leq n$. We may omit the subscripts $p_1, \ldots, p_n$ when they are clear from the context. The *domain* and *range* of a *substitution* σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively; a substitution σ will be denoted by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ if $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = t_i$ for all $1 \leq i \leq n$. The application $\sigma(t)$ of substitution σ to term t is abbreviated to $t\sigma$.

A set of rewrite rules $l \to r$ such that l is a nonvariable term and r is a term whose variables appear in l is called a *term rewriting system* (TRS for short); terms l and r are called the left-hand side and the right-hand side of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}_\mathcal{R}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C}_\mathcal{R} = \mathcal{F} \setminus \mathcal{D}_\mathcal{R}$. *Constructor terms* of $\mathcal{R}$ are terms over $\mathcal{C}_\mathcal{R}$ and $\mathcal{V}$. Ground (i.e., without variables) terms and ground constructor terms are denoted by $\mathcal{T}(\mathcal{F})$ and $\mathcal{T}(\mathcal{C}_\mathcal{R})$, respectively. We sometimes omit $\mathcal{R}$ from $\mathcal{D}_\mathcal{R}$ and $\mathcal{C}_\mathcal{R}$ if it is clear from the context. A substitution σ is a *constructor substitution* if $x\sigma \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ for all variables x. $\mathcal{R}$ is a *constructor system* if the left-hand sides of its rules have the form $f(s_1, \ldots, s_n)$ where s_i are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$, for all $i = 1, \ldots, n$.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as follows: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exists a position p in s, a rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ with $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is often denoted by $s \to_{p,l\to r} t$ to make explicit the position and rule used in this step. Moreover, if no proper subterms of $s|_p$ are reducible, then we speak of an *innermost* reduction step, denoted by $s \xrightarrow{i}_\mathcal{R} t$. The instantiated left-hand side $l\sigma$ is called a *redex*. A term t is called *irreducible* or in *normal form* w.r.t. a TRS $\mathcal{R}$ if there is no term s with $t \to_\mathcal{R} s$. We denote the set of normal forms by $NF_{\mathcal{R}}$.

A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation $\to$, we denote by $\to^*$ its reflexive and transitive closure. Thus $t \to^*_{\mathcal{R}} s$ means that t can be reduced to s in $\mathcal{R}$ in zero or more steps; we also use $t \to^n_{\mathcal{R}} s$ to denote that t can be reduced to s in exactly n rewrite steps.

3. Conversion to Tail Recursive Form

In this section, we introduce an automatic conversion to tail recursive form and prove its correctness.

3.1. The Transformation

As illustrated in the previous section, the basic idea consists in introducing two fresh constructor functions, id and cont, to represent *contexts*. In the following definition, for the sake of simplicity, we consider that the function to be inverted is *self-recursive*.

Definition 1 (tail recursive transformation). Let $\mathcal{R}$ be a TRS and let f/n be a function symbol defined by the rules $R_f \subseteq \mathcal{R}$. The TRS $\mathcal{T}ail(\mathcal{R}, f)$ obtained from $\mathcal{R}$ by transforming the rules of function f to tail recursive form is given by

$$\begin{array}{rl} \mathcal{T}ail(\mathcal{R}, f) = & (\mathcal{R} \setminus R_f) \\ & \cup \; \{f(\overline{x_n}) \to f_{tail}(\overline{x_n}, \mathsf{id}),\; \mathsf{eval}(\mathsf{id}, x) \to x\} \\ & \cup \; \bigcup_{l \to r \in R_f} tail_f(l \to r) \end{array}$$

where the auxiliary function $tail_f$ is defined as follows:

- If $l = f(\overline{t_n})$ and r does not contain calls to function f, then

$$tail_f(l \to r) = \{f_{tail}(\overline{t_n}, k) \to \mathsf{eval}(k, r)\}$$

- If r contains at least one call to function f—if there are several calls, we select any call, e.g., the leftmost innermost one—we proceed as follows. Let $l = f(\overline{t_n})$ and $r = C[f(\overline{s_n})]$. Then, we have

$$tail_f(l \to r) = \left\{ \begin{array}{rcl} f_{tail}(\overline{t_n}, k) & \to & f_{tail}(\overline{s_n}, \mathsf{cont}(k, \overline{y_m})) \\ \mathsf{eval}(\mathsf{cont}(k, \overline{y_m}), w) & \to & \mathsf{eval}(k, C[w]) \end{array} \right\}$$

  where cont is a fresh constructor symbol and $\overline{y_m}$ are the variables of $C[\,]$. Note that, when $C[\,]$ is $\Box$, we have

$$tail_f(l \to r) = \{f_{tail}(\overline{t_n}, k) \to f_{tail}(\overline{s_n}, k)\}$$

Example 2. Let us consider the Fibonacci program $\mathcal{R}_{\mathit{fib}}$:

$$\begin{array}{rcl}
\mathsf{fib}(\mathsf{0}) & \to & \mathsf{0} \\
\mathsf{fib}(\mathsf{s}(\mathsf{0})) & \to & \mathsf{s}(\mathsf{0}) \\
\mathsf{fib}(\mathsf{s}(\mathsf{s}(n))) & \to & \mathsf{add}(\mathsf{fib}(\mathsf{s}(n)), \mathsf{fib}(n)) \\
\mathsf{add}(\mathsf{0}, y) & \to & y \\
\mathsf{add}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{add}(x, y))
\end{array}$$

where natural numbers are built from $\mathsf{0}$ and s (successor) functions. Transforming function fib to tail recursion, we get $\mathcal{R}'_{\mathit{fib}} = \mathcal{T}\mathit{ail}(\mathcal{R}_{\mathit{fib}}, \mathsf{fib})$:

$$\begin{array}{rcl}
\mathsf{fib}(n) & \to & \mathsf{fib_{tail}}(n, \mathsf{id}) \\
\mathsf{fib_{tail}}(\mathsf{0}, k) & \to & \mathsf{eval}(k, \mathsf{0}) \\
\mathsf{fib_{tail}}(\mathsf{s}(\mathsf{0}), k) & \to & \mathsf{eval}(\mathsf{s}(\mathsf{0}, k)) \\
\mathsf{fib_{tail}}(\mathsf{s}(\mathsf{s}(n)), k) & \to & \mathsf{fib_{tail}}(\mathsf{s}(n), \mathsf{cont}(k, n)) \\
\mathsf{eval}(\mathsf{id}, x) & \to & x \\
\mathsf{eval}(\mathsf{cont}(k, x), w) & \to & \mathsf{add}(w, \mathsf{fib}(x))
\end{array}$$

together with the original definition of the function add. Moreover, we could also transform function add to tail recursive form, thus obtaining $\mathcal{R}''_{\mathit{fib}} = \mathcal{T}\mathit{ail}(\mathcal{R}'_{\mathit{fib}}, \mathsf{add})$:

$$\begin{array}{rcl}
\mathsf{add}(x, y) & \to & \mathsf{add_{tail}}(x, y, \mathsf{id_{add}}) \\
\mathsf{add_{tail}}(\mathsf{0}, y, k) & \to & \mathsf{eval_{add}}(k, y) \\
\mathsf{add_{tail}}(\mathsf{s}(x), y, k) & \to & \mathsf{add_{tail}}(x, y, \mathsf{cont_{add}}(k)) \\
\mathsf{eval_{add}}(\mathsf{id_{add}}, x) & \to & x \\
\mathsf{eval_{add}}(\mathsf{cont_{add}}(k), w) & \to & \mathsf{s}(\mathsf{eval_{add}}(k, w))
\end{array}$$

together with the previous definitions for fib, $\mathsf{fib_{tail}}$ and eval.

Although Definition 1 only considers self-recursive functions, its extension to deal with mutually recursive functions is not difficult. In particular, one can consider a *transformational* approach that proceeds as follows. Let F be a set of mutually recursive functions. Then, we introduce a fresh function symbol g, a fresh dummy constant $\perp$, and fresh constants c_f for all $f \in F$. Now, we can transform the mutual recursion into a direct self-recursion, e.g., as follows: every term $f(\bar{t})$ is replaced by $g(\mathsf{c}_f, \bar{t}, \perp, \ldots, \perp)$, where the arity of the fresh function g is the maximum arity of the functions in F, c_f is a constructor constant that identifies function f, and $\perp$ is used to fill the arguments which are not needed (when the arity of f is smaller than the arity of g). Finally, one can apply $\mathcal{T}ail$ to the self-recursive case, and again replace "$g(\mathsf{c}_f,$" by "$f_{tail}($" and also drop $\perp$ from the rules. Let us illustrate this idea with an example.

Example 3. Consider the following contrieved TRS to double a given natural number:

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathsf{double}(n) & \to & \mathsf{f}(n, \mathsf{0}) \\ \mathsf{f}(\mathsf{0}, y) & \to & y \\ \mathsf{f}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{s}(\mathsf{g}(x, y, \mathsf{0}))) \\ \mathsf{g}(\mathsf{0}, y, z) & \to & y \\ \mathsf{g}(\mathsf{s}(x), y, z) & \to & \mathsf{s}(\mathsf{f}(x, \mathsf{s}(y))) \end{array} \right\}$$

A direct extension of $\mathcal{T}ail$ to deal with mutual recursion would transform $\mathcal{R}$ into the following TRS:

$$\mathcal{T}ail(\mathcal{R}, \{\mathsf{f}, \mathsf{g}\}) = \left\{ \begin{array}{rcl} \mathsf{double}(n) & \to & \mathsf{f}(n, \mathsf{0}) \\ \mathsf{f}(n, y) & \to & \mathsf{f}_{tail}(n, y, \mathsf{id}) \\ \mathsf{f}_{tail}(\mathsf{0}, y, k) & \to & \mathsf{eval}(k, y) \\ \mathsf{f}_{tail}(\mathsf{s}(x), y, k) & \to & \mathsf{g}_{tail}(x, y, \mathsf{0}, \mathsf{cont}_\mathsf{f}(k)) \\ \mathsf{g}(n, y, z) & \to & \mathsf{g}_{tail}(n, y, z, \mathsf{id}) \\ \mathsf{g}_{tail}(\mathsf{0}, y, z, k) & \to & \mathsf{eval}(k, y) \\ \mathsf{g}_{tail}(\mathsf{s}(x), y, z, k) & \to & \mathsf{f}_{tail}(x, \mathsf{s}(y), \mathsf{cont}_\mathsf{g}(k)) \\ \mathsf{eval}(\mathsf{id}, y) & \to & y \\ \mathsf{eval}(\mathsf{cont}_\mathsf{f}(k), y) & \to & \mathsf{eval}(k, \mathsf{s}(\mathsf{s}(y))) \\ \mathsf{eval}(\mathsf{cont}_\mathsf{g}(k), y) & \to & \mathsf{eval}(k, \mathsf{s}(y)) \end{array} \right\}$$

Now, let us show how the transformational approach sketched above works. First, we introduce a fresh function h and constructor constants c_f and c_g. Then, the mutual recursion is transformed to a self-recursion as follows:

$$\mathcal{R}' = \left\{ \begin{array}{rcl} \mathsf{double}(n) & \to & \mathsf{h}(\mathsf{fun}_\mathsf{f}, n, \mathsf{0}, \perp) \\ \mathsf{h}(\mathsf{c}_\mathsf{f}, \mathsf{0}, y, \perp) & \to & y \\ \mathsf{h}(\mathsf{c}_\mathsf{f}, \mathsf{s}(x), y, \perp) & \to & \mathsf{s}(\mathsf{s}(\mathsf{h}(\mathsf{c}_\mathsf{g}, x, y, \mathsf{0}))) \\ \mathsf{h}(\mathsf{c}_\mathsf{g}, \mathsf{0}, y, z) & \to & y \\ \mathsf{h}(\mathsf{c}_\mathsf{g}, \mathsf{s}(x), y, z) & \to & \mathsf{s}(\mathsf{h}(\mathsf{c}_\mathsf{f}, x, \mathsf{s}(y), \perp)) \end{array} \right\}$$

By applying function $\mathcal{T}ail$ to $\mathcal{R}'$, we get the following TRS:

$$\mathcal{T}ail(\mathcal{R}', \mathsf{h}) = \left\{ \begin{array}{rcl} \mathsf{double}(n) & \to & \mathsf{h}(\mathsf{fun_f}, n, \mathsf{0}, \bot) \\ \mathsf{h}(x, y, z, w) & \to & \mathsf{h}_{tail}(x, y, z, w, \mathsf{id}) \\ \mathsf{h}_{tail}(\mathsf{c_f}, \mathsf{0}, y, \bot, k) & \to & \mathsf{eval}(k, y) \\ \mathsf{h}_{tail}(\mathsf{c_f}, \mathsf{s}(x), y, \bot, k) & \to & \mathsf{h}_{tail}(\mathsf{c_g}, x, y, \mathsf{0}, \mathsf{cont_f}(k)) \\ \mathsf{h}_{tail}(\mathsf{c_g}, \mathsf{0}, y, z, k) & \to & \mathsf{eval}(k, y) \\ \mathsf{h}_{tail}(\mathsf{c_g}, \mathsf{s}(x), y, z, k) & \to & \mathsf{h}_{tail}(\mathsf{c_f}, x, \mathsf{s}(y), \bot, \mathsf{cont_g}(k)) \\ \mathsf{eval}(\mathsf{id}, y) & \to & y \\ \mathsf{eval}(\mathsf{cont_f}(k), y) & \to & \mathsf{eval}(k, \mathsf{s}(\mathsf{s}(y))) \\ \mathsf{eval}(\mathsf{cont_g}(k), y) & \to & \mathsf{eval}(k, \mathsf{s}(y)) \end{array} \right\}$$

Finally, by replacing "$\mathsf{h}(\mathsf{c_f},$" and "$\mathsf{h}(\mathsf{c_g},$" with "$\mathsf{f}($" and "$\mathsf{g}($", resp., and by removing all occurrences of $\bot$ from the rules, the system $\mathcal{T}ail(\mathcal{R}', \mathsf{h})$ is transformed into (a simplified variant of) $\mathcal{T}ail(\mathcal{R}, \{\mathsf{f}, \mathsf{g}\})$.

3.2. Correctness

Let us now discuss the correctness of $\mathcal{T}ail$. Unfortunately, the function $\mathcal{T}ail$ does not in general preserve innermost reduction sequences, even if we restrict to those that end with a constructor term (a *value*).

Example 4. Consider the following TRS:

$$\mathcal{R} = \left\{ \begin{array}{rcl} \mathsf{f}(x) & \to & \mathsf{g}(\mathsf{h}(x)) \\ \mathsf{g}(\mathsf{s}(x)) & \to & \mathsf{0} \\ \mathsf{h}(\mathsf{0}) & \to & \mathsf{0} \\ \mathsf{h}(\mathsf{s}(\mathsf{s}(x))) & \to & \mathsf{s}(\mathsf{h}(x)) \end{array} \right\}$$

Note that $\mathcal{R}$ is not sufficiently complete.[2] Consider, e.g., the normal form

[2]An n-ary function symbol $f \in \mathcal{D}_\mathcal{R}$ is called *sufficiently complete w.r.t.* $\mathcal{R}$ if for all ground constructor terms $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}_\mathcal{R})$, there exists a ground constructor term $t \in \mathcal{T}(\mathcal{C}_\mathcal{R})$ such that $f(t_1, \ldots, t_n) \to^+_\mathcal{R} t$. $\mathcal{R}$ is called *sufficiently complete* if every defined symbol $f \in \mathcal{D}_\mathcal{R}$ is sufficiently complete w.r.t. $\mathcal{R}$. Note that for a terminating TRS $\mathcal{R}$, $\mathcal{R}$ is sufficiently complete iff $NF_\mathcal{R} \cap \mathcal{T}(\mathcal{F}) = \mathcal{T}(\mathcal{C}_\mathcal{R})$.

$\mathsf{h}(\mathsf{s}(\mathsf{0}))$. Here, $\mathcal{R}$ is transformed by $\mathcal{T}ail$ as follows:

$$\mathcal{T}ail(\mathcal{R},\mathsf{h}) = \left\{ \begin{array}{rcl} \mathsf{f}(x) & \to & \mathsf{g}(\mathsf{h}(x)) \\ \mathsf{g}(\mathsf{s}(x)) & \to & \mathsf{0} \\ \mathsf{h}(x) & \to & \mathsf{h}_{tail}(x, \mathsf{id}) \\ \mathsf{h}_{tail}(\mathsf{0}, k) & \to & \mathsf{eval}(k, \mathsf{0}) \\ \mathsf{h}_{tail}(\mathsf{s}(\mathsf{s}(x)), k) & \to & \mathsf{h}_{tail}(x, \mathsf{cont}(k)) \\ \mathsf{eval}(\mathsf{id}, x) & \to & x \\ \mathsf{eval}(\mathsf{cont}(k), x) & \to & \mathsf{eval}(k, \mathsf{s}(x)) \end{array} \right\}$$

Now, we have the derivation

$$\mathsf{f}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \overset{i}{\to}_{\mathcal{R}} \mathsf{g}(\mathsf{h}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \overset{i}{\to}_{\mathcal{R}} \mathsf{g}(\mathsf{s}(\mathsf{h}(\mathsf{s}(\mathsf{0})))) \overset{i}{\to}_{\mathcal{R}} \mathsf{0}$$

but this derivation is not possible in $\mathcal{R}'$:

$$\begin{array}{l} \mathsf{f}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))) \overset{i}{\to}_{\mathcal{T}ail(\mathcal{R},\mathsf{h})} \mathsf{g}(\mathsf{h}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) \overset{i}{\to}_{\mathcal{T}ail(\mathcal{R},\mathsf{h})} \mathsf{g}(\mathsf{h}_{tail}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))), \mathsf{id})) \\ \quad \overset{i}{\to}_{\mathcal{T}ail(\mathcal{R},\mathsf{h})} \mathsf{g}(\mathsf{h}_{tail}(\mathsf{s}(\mathsf{0}), \mathsf{cont}(\mathsf{id}))) \overset{i}{\not\to}_{\mathcal{T}ail(\mathcal{R},\mathsf{h})} \end{array}$$

Intuitively speaking, the problem with tail recursion functions is that they do not preserve the semantics when only *partial* values—terms rooted by some constructor symbols, like $\mathsf{s}(\mathsf{h}(\mathsf{s}(\mathsf{0})))$ above—were required in the original program, since tail recursive functions either terminate producing a *total* value—a constructor term—or an expression rooted by a defined function symbol, like $\mathsf{h}_{tail}(\mathsf{s}(\mathsf{0}), \mathsf{cont}(\mathsf{id}))$ above. A similar situation occurs with continuations and lazy functional languages like Haskell, where the introduction of continuations may force the eager evaluation of some expression, thus changing the original semantics.[3]

Therefore, in the following, we further restrict the intended semantics to only consider so called *constructor-based* reductions [8], a particular case of innermost reduction where only constructor matchers are allowed. Formally, an innermost reduction step $s \overset{i}{\to}_{\mathcal{R}} t$ is said constructor-based if all the proper subterms of the selected redex $s|_p$ are constructor terms, which is denoted by $s \underset{\mathsf{c}}{\to}_{\mathcal{R}} t$. There are classes of rewrite systems for which $\overset{i}{\to}_{\mathcal{R}} = \underset{\mathsf{c}}{\to}_{\mathcal{R}}$, e.g., for

[3]Consider, e.g., the functions app and $\mathsf{app_c}$ shown in Section 1. Given a non-terminating function $\perp$ (defined by $\perp \to \perp$) and the well-known function head that returns the head of a list, we have that $\mathsf{app}\ (1 : \perp)\ \perp$ reduces to 1 in Haskell, while $\mathsf{app_c}\ (1 : \perp)\ \perp$ is undefined.

sufficiently complete systems or non-erasing systems; nevertheless, we prefer to require constructor-based reductions in our results in order to keep them more general.

Now, we prove the correctness of the $\mathcal{T}ail$ transformation. Here, we make some assumptions to simplify the proof of correctness. In particular, we assume that the considered function, f, is unary and is defined by means of a single non-recursive rule and a single recursive rule as follows:

$$\begin{array}{rcll} f(a) & \to & r & (R_1) \\ f(b) & \to & C[f(s)] & (R_2) \end{array}$$

where r, s are constructor terms and $C[\ ]$ is a constructor context, i.e., we assume that f is not only self-recursive but also linear (which means that there is just one recursive call in the right-hand side of the recursive rule).

Therefore, we assume that $\mathcal{R}' = \mathcal{T}ail(\mathcal{R}, f)$, the output of the tail recursive conversion for function f, contains the rules

$$\begin{array}{rcll} f(x) & \to & f_{tail}(x, \mathsf{id}) & \\ f_{tail}(a, k) & \to & \mathsf{eval}(k, r) & (R_1') \\ f_{tail}(b, k) & \to & f_{tail}(s, \mathsf{cont}(k, \overline{y_m})) & (R_2') \\ \mathsf{eval}(\mathsf{id}, w) & \to & w & \\ \mathsf{eval}(\mathsf{cont}(k, \overline{y_m}), w) & \to & \mathsf{eval}(k, C[w]) & \end{array}$$

Extending our results for arbitrary systems is not difficult but makes the proofs much less intuitive and technically more involved (see below).

Our first lemma shows some basic equivalence between the reductions in the original system and in the tail recursive one.

Lemma 5. *Let $\mathcal{R}$ be a constructor TRS and let $\mathcal{R}' = \mathcal{T}ail(\mathcal{R}, f)$. Then,*

$$f(t_0) \underset{\mathsf{c}}{\to}{}^{n}_{\mathcal{R}} C\sigma_1[\ldots C\sigma_n[f(t_n)]\ldots]$$

iff

$$f_{tail}(t_0, w) \underset{\mathsf{c}}{\to}{}^{n}_{\mathcal{R}'} f_{tail}(t_n, \mathsf{cont}(\ldots \mathsf{cont}(w, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n))$$

where $\overline{y_m}$ are the variables of $C[]$ and $\sigma_1, \ldots, \sigma_n$ are constructor substitutions.

Proof. We prove the claim by induction on the number $n \geq 0$ of reduction steps. The base case $n = 0$ follows trivially since $C[\,]$ is the empty context.

So we proceed with the inductive case $n > 0$. Here, we assume that the former derivation has the following form:

$$f(t_0) \underset{\mathsf{c}}{\rightarrow}_{R_2} C[f(s)]\sigma_1 = C\sigma_1[f(t_1)] \underset{\mathsf{c}}{\rightarrow}^{n-1}_{\mathcal{R}} C\sigma_1[\ldots C\sigma_n[f(t_n)]\ldots]$$

where $b\sigma_i = t_{i-1}$ and $s\sigma_i = t_i$, for all $i = 1,\ldots,n$. Thus $f(t_0) \underset{\mathsf{c}}{\rightarrow}_{R_2} C\sigma_1[f(t_1)]$ iff $f_{tail}(t_0, w) \underset{\mathsf{c}}{\rightarrow}_{R'_2} f_{tail}(s, \mathsf{cont}(w, \overline{y_m}))\sigma_1 = f_{tail}(t_1, \mathsf{cont}(w, \overline{y_m}\sigma_1))$.[4] Hence the claim follows by applying the induction hypothesis. □

Our next auxiliary lemma states a useful property of tail recursive systems.

Lemma 6. *Let $\mathcal{R}$ be a constructor TRS and let $\mathcal{R}' = \mathcal{T}ail(\mathcal{R}, f)$. Then,*

$$\mathsf{eval}(\mathsf{cont}(\ldots\mathsf{cont}(w, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n), t) \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}'} \mathsf{eval}(w, C\sigma_1[\ldots C\sigma_n[t]\ldots])$$

where $\overline{y_m}$ are the variables of $C[]$, $\overline{y_m} \notin \mathcal{V}ar(t)$, and $\sigma_1,\ldots,\sigma_n$ are constructor substitutions.

Proof. Trivial by definition of eval in $\mathcal{R}'$. □

Now, we can proceed with the proof of our main result:

Theorem 7. *Let $\mathcal{R}$ be a constructor TRS and let $\mathcal{R}' = \mathcal{T}ail(\mathcal{R}, f)$ be the output of the tail recursive conversion for some function f. Then,*

- *$\mathcal{R}'$ is a constructor TRS and*
- *$f(t_0) \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} t$ iff $f(t_0) \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}'} t$ for constructor terms t_0, t.*

Proof. The fact that $\mathcal{R}'$ is a constructor TRS is a trivial consequence from the fact that $\mathcal{R}$ is a constructor TRS and the way in which the left-hand sides are modified: only single variable arguments are added to f and both id and cont are constructor symbols.

($\Rightarrow$) Let us consider that $f(t_0) \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} t$ has the form

$$f(t_0) \underset{\mathsf{c}}{\rightarrow}^{n}_{\mathcal{R}} C\sigma_1[\ldots C\sigma_n[f(t_n)]\ldots] \underset{\mathsf{c}}{\rightarrow}_{\mathcal{R}} C\sigma_1[\ldots C\sigma_n[r\sigma_{n+1}]\ldots] = t$$

[4]Here, we use the fact that $k \notin \mathcal{V}ar(b) \cup \mathcal{V}ar(s)$ by construction.

where $b\sigma_i = t_{i-1}$ and $s\sigma_i = t_i$ for all $i = 1, \ldots, n$, and $a\sigma_{n+1} = t_n$. Note that $r\sigma_{n+1}$ is a constructor term by construction (since we assumed r to be a constructor term and all σ_i are constructor substitutions).

Therefore, in $\mathcal{R}'$, we have $f(t_0) \xrightarrow[\mathsf{c}]{}_{\mathcal{R}'} f_{tail}(t_0, \mathsf{id})$. By Lemma 5, we have $f_{tail}(t_0, \mathsf{id}) \xrightarrow[\mathsf{c}]{}{}^{n}_{\mathcal{R}'} f_{tail}(t_n, \mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n))$. Moreover, since $a\sigma_{n+1} = t_n$, we have

$$\begin{array}{l} f_{tail}(t_n, \mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n)) \\ \qquad\qquad \xrightarrow[\mathsf{c}]{}_{\mathcal{R}'} \mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n), r\sigma_{n+1}) \end{array}$$

Now, by Lemma 6, we have

$$\begin{array}{r} \mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n), r\sigma_{n+1}) \\ \xrightarrow[\mathsf{c}]{}{}^{*}_{\mathcal{R}'} \mathsf{eval}(\mathsf{id}, C\sigma_1[\ldots C\sigma_n[r\sigma_{n+1}] \ldots]) \end{array}$$

And, finally, by applying the base case of eval we get

$$\mathsf{eval}(\mathsf{id}, C\sigma_1[\ldots C\sigma_n[r\sigma_{n+1}] \ldots]) \xrightarrow[\mathsf{c}]{}_{\mathcal{R}'} C\sigma_1[\ldots C\sigma_n[r\sigma_{n+1}] \ldots] = t$$

($\Leftarrow$) The proof is analogous by applying Lemma 5 and 6 in the opposite direction. $\square$

Now, we discuss how the assumptions made in the proof scheme above can be relaxed:

1. On the first hand, we can consider arbitrary, mutually recursive functions by transforming them into a self-recursive function as discussed above, so this is not relevant for proving correctness.
2. Considering functions with an arbitrary number of arguments is not relevant too, since we can just group all arguments by introducing a fresh constructor symbol (i.e., a tuple symbol).
3. Extending the proofs for dealing with several non-recursive and several recursive rules is tedious but easy. To be more precise, one should extend the previous results to consider different contexts $C_1, \ldots, C_j$ (for the right-hand sides of the recursive rules), together with the associated continuation constructors, $\mathsf{cont}_1, \ldots, \mathsf{cont}_j$, $j > 0$.
4. Assuming that the right-hand side of the non-recursive rules is an arbitrary term (rather than a constructor term) only affects to the proof of Theorem 7. The extension would be straightforward by introducing some intermediate derivations which are the same in both the original and transformed systems.

5. Finally, the main difficulty comes from the extension to deal with arbitrary, non-linear recursive rules, i.e., from considering that the right-hand sides of the recursive rules may contain calls to other functions. This extension has a strong influence on the evaluation order and makes the correspondence between the derivations in the original and transformed systems more difficult to establish.

Now, we extend the correctness result in order to overcome the limitations mentioned in points (4) and (5) above, the most important ones. In the following, given a rewrite sequence $t_0 \underset{\mathsf{c}}{\rightarrow}_{p_0,\mathcal{R}} t_1 \underset{\mathsf{c}}{\rightarrow}_{p_1,\mathcal{R}} \cdots \underset{\mathsf{c}}{\rightarrow}_{p_{n-1},\mathcal{R}} t_n$, we write $t_1 \underset{\mathsf{c}}{\rightarrow}^{n}_{p\leq,\mathcal{R}} t_n$ (resp. $t_1 \underset{\mathsf{c}}{\rightarrow}^{n}_{p<,\mathcal{R}} t_n$) if $p \leq p_i$ (resp. $p < p_i$) for all $i = 0, \ldots, n$. Moreover, by the definition of constructor reduction $\underset{\mathsf{c}}{\rightarrow}$, we have the following properties:

- no redex w.r.t. $\underset{\mathsf{c}}{\rightarrow}$ contain a redex as its proper subterm, and
- the positions of all redexes appearing in a term are disjoint, i.e., for all redex positions p, p', we have neither $p < p'$ nor $p' < p$.

Due to these properties, rewrite steps over disjoint positions can be exchanged while preserving the computed normal form and, thus, the following standardization lemma trivially holds:

Lemma 8. *Let $\mathcal{R}$ be a constructor TRS, $C[\,]_p$ a context, f/n a defined symbol, $\overline{s_n}$ a sequence of n terms, and u a constructor term. If $C[f(\overline{s_n})]_p \underset{\mathsf{c}}{\rightarrow}^{*}_{p,\mathcal{R}} u$ with k steps, then there exist a sequence $\overline{t_n}$ of n constructor terms and a constructor term u' such that $C[f(\overline{s_n})]_p \underset{\mathsf{c}}{\rightarrow}^{*}_{p<,\mathcal{R}} C[f(\overline{t_n})]_p \underset{\mathsf{c}}{\rightarrow}^{*}_{p\leq,\mathcal{R}} C[u']_p \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} u$ with k steps.*

Now, we generalize Lemma 6 in order to overcome the limitation mentioned in point (5) above:

Lemma 9. *Let $\mathcal{R}$ be a constructor TRS and $\mathcal{R}' = \mathcal{T}ail(\mathcal{R}, f)$. Let $C[\,]$ be a context, $u_0, \ldots, u_n$ constructor terms, and $\sigma_1, \ldots, \sigma_n$ constructor substitu-*

tions. If $C\sigma_i[u_i] \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}'} u_{i-1}$ *for all* $1 \leq i \leq n$, *then*

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(w, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n), u_n)} \\
\underset{\mathsf{c}}{\rightarrow}_{\mathcal{R}'} & \mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(w, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_{n-1}), C\sigma_n[u_n]) \\
\underset{\mathsf{c}}{\rightarrow}^{*}_{\varepsilon<,\mathcal{R}'} & \mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(w, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_{n-1}), u_{n-1}) \\
\underset{\mathsf{c}}{\rightarrow}_{\mathcal{R}'} & \mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(w, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_{n-2}), C\sigma_{n-1}[u_{n-1}]) \\
\underset{\mathsf{c}}{\rightarrow}^{*}_{\varepsilon<,\mathcal{R}'} & \ldots \\
\underset{\mathsf{c}}{\rightarrow}^{*}_{\varepsilon<,\mathcal{R}'} & \mathsf{eval}(\mathsf{cont}(w, \overline{y_m}\sigma_1), u_1) \\
\underset{\mathsf{c}}{\rightarrow}_{\mathcal{R}'} & \mathsf{eval}(w, C\sigma_1[u_1]) \\
\underset{\mathsf{c}}{\rightarrow}^{*}_{\varepsilon<,\mathcal{R}'} & \mathsf{eval}(w, u_0)
\end{array}$$

for all continuation term w, *where* $\overline{y_m}$ *are the variables of* $C[\,]$, $\overline{y_m} \not\in \mathcal{V}ar(t)$.

Proof. Trivial by definition of eval in $\mathcal{R}'$. □

Then, we can state an extension of Theorem 7 that overcomes the limitations mentioned in points (4) and (5) above as follows:

Theorem 10. *Let* $\mathcal{R}$ *be a constructor TRS and let* $\mathcal{R}' = \mathcal{T}ail(\mathcal{R}, f)$ *be the output of the tail recursive conversion for some function* f. *Then,*

- $\mathcal{R}'$ *is a constructor TRS and*
- $s \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} t$ *iff* $s \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}'} t$, *where* s *is a term over the original signature and* t *is a constructor term.*

Proof. As for Theorem 7, the fact that $\mathcal{R}'$ is a constructor TRS is a trivial consequence from the fact that $\mathcal{R}$ is a constructor TRS and the way in which the left-hand sides are modified: only single variable arguments are added to f and both id and cont are constructor symbols.

($\Rightarrow$) We prove this direction by induction on the length of $s \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} t$. Since the case when s is not rooted by f can easily be proved, we only consider the remaining case. Thanks to Lemma 8, we can consider that $s \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} t$ has the form

$$\begin{array}{llllll}
s = f(t_0) & \underset{\mathsf{c}}{\rightarrow}^{*}_{\varepsilon,\mathcal{R}} & C\sigma_1[f(s_1)]_p & \underset{\mathsf{c}}{\rightarrow}^{*}_{p<,\mathcal{R}} & C\sigma_1[f(t_1)]_p \\
 & \underset{\mathsf{c}}{\rightarrow}_{p,\mathcal{R}} & C\sigma_1[C\sigma_2[f(s_2)]_p]_p & \underset{\mathsf{c}}{\rightarrow}^{*}_{p.p<,\mathcal{R}} & \ldots \\
 & \underset{\mathsf{c}}{\rightarrow}^{*}_{p\ldots p,\mathcal{R}} & C\sigma_1[\ldots C\sigma_n[f(s_n)]_p \ldots]_p & \underset{\mathsf{c}}{\rightarrow}^{*}_{p\ldots p<,\mathcal{R}} & C\sigma_1[\ldots C\sigma_n[f(t_n)]_p \ldots]_p \\
 & \underset{\mathsf{c}}{\rightarrow}_{p\ldots p,\mathcal{R}} & C\sigma_1[\ldots C\sigma_n[r\sigma]_p \ldots]_p \\
 & \underset{\mathsf{c}}{\rightarrow}^{*}_{p\ldots p\leq,\mathcal{R}} & C\sigma_1[\ldots C\sigma_n[u_n]_p \ldots]_p \\
 & \underset{\mathsf{c}}{\rightarrow}^{*}_{p\ldots p\leq,\mathcal{R}} & C\sigma_1[\ldots C\sigma_{n-1}[u_{n-1}]_p \ldots]_p \\
 & \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} & \ldots \\
 & \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} & C\sigma_1[u_1] \quad \underset{\mathsf{c}}{\rightarrow}^{*}_{\mathcal{R}} \quad t
\end{array}$$

where $\sigma_1, \ldots, \sigma_n, \sigma$ are constructor substitutions, $u_1, \ldots, u_n$ are constructor terms, and $b\sigma_i = t_{i+1}$ and $s\sigma_i = s_i$ for all $i = 1, \ldots, n$, and $a\sigma = t_n$. By the induction hypothesis, we have

- $s_i \xrightarrow[\mathsf{c}]{}^*_{\mathcal{R}'} t_i$ for all $i = 1, \ldots, n$,
- $r\sigma \xrightarrow[\mathsf{c}]{}^*_{\mathcal{R}'} u_n$,
- $C\sigma_i[u_i] \xrightarrow[\mathsf{c}]{}^*_{\mathcal{R}'} u_{i-1}$ for all $i = 2, \ldots, n$, and
- $C\sigma_1[u_1] \xrightarrow[\mathsf{c}]{}^*_{\mathcal{R}'} t$.

By definition, in $\mathcal{R}'$, we have $f(t_0) \xrightarrow[\mathsf{c}]{}_{\mathcal{R}'} f_{tail}(t_0, \mathsf{id})$. By Lemma 5, we have $f_{tail}(t_0, \mathsf{id}) \xrightarrow[\mathsf{c}]{}^*_{\mathcal{R}'} f_{tail}(t_n, \mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n))$. Moreover, since $a\sigma = t_n$, we have

$$\begin{array}{l} f_{tail}(t_n, \mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n)) \\ \qquad\qquad \xrightarrow[\mathsf{c}]{}_{\mathcal{R}'} \mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n), r\sigma) \end{array}$$

Now, by Lemma 9, we have

$$\mathsf{eval}(\mathsf{cont}(\ldots \mathsf{cont}(\mathsf{id}, \overline{y_m}\sigma_1), \ldots \overline{y_m}\sigma_n), r\sigma) \xrightarrow[\mathsf{c}]{}^*_{\mathcal{R}'} \mathsf{eval}(\mathsf{id}, t)$$

And, finally, by applying the base case of eval we get

$$\mathsf{eval}(\mathsf{id}, t) \xrightarrow[\mathsf{c}]{}_{\mathcal{R}'} t$$

($\Leftarrow$) The proof is analogous by applying Lemma 5 and 9 in the opposite direction. $\square$

3.3. Well-typedness

Finally, we conclude this section by observing that, if $\mathcal{R}$ is a well-typed TRS, so is $\mathcal{T}ail(\mathcal{R}, f)$. This is an easy consequence of the following facts:

- Let us consider that f is defined in $\mathcal{R}$ by the following rules:

$$\begin{array}{rcl} f(\overline{c_n}) & \to & r \\ f(\overline{t_n}) & \to & C[f(\overline{s_n})] \\ & \ldots & \\ f(\overline{t'_n}) & \to & C'[f(\overline{s'_n})] \end{array}$$

Here, we should consider a new type of the following form:

$$\mathsf{data}\ \mathsf{C} \ = \ \mathsf{id} \mid \mathsf{cont}_1(\mathsf{C},\mathsf{D}_1,\ldots,\mathsf{D}_m) \mid \ \ldots \ \mid \mathsf{cont}_k(\mathsf{C},\mathsf{D}'_1,\ldots,\mathsf{D}'_j)$$

where id and $\mathsf{cont}_1,\ldots,\mathsf{cont}_k$ are fresh constructor symbols, $\overline{\mathsf{D}_m}$ are the types of the variables $\overline{y_m}$ in $C[\,]$, and $\overline{\mathsf{D}'_j}$ are the types of the variables $\overline{y'_j}$ in $C'[\,]$.

- Assuming that the type of the function f is $\mathsf{T_1} \times \ldots \times \mathsf{T_n} \to \mathsf{T}$, the types of the new functions introduced by the transformation are the following:

$$\begin{array}{lcl} f_{tail} & :: & \mathsf{T_1} \times \ldots \times \mathsf{T_n} \times \mathsf{C} \to \mathsf{T} \\ \mathsf{eval} & :: & \mathsf{C} \times \mathsf{T} \to \mathsf{T} \end{array}$$

- Now, one can prove that the application of all of the new rules introduced by the transformation preserve well-typed terms:
  - For the initial rules, $f(\overline{x_n}) \to f_{tail}(x_n,\mathsf{id})$ and $\mathsf{eval}(\mathsf{id},x) \to x$, the claim follows trivially.
  - Consider now a rule of the form $f_{\mathsf{tail}}(\overline{t_n},k) \to \mathsf{eval}(k,r)$. By assuming that the term $f_{tail}(\overline{t_n},k)$ is well-typed, we have that k has type C. Therefore, $\mathsf{eval}(k,r)$ is also well-typed when r is well-typed in the original TRS.
  - Consider a rule of the form $f_{tail}(\overline{t_n},k) \to f_{tail}(\overline{s_n},\mathsf{cont}_1(k,\overline{y_m}))$. By assuming that the term $f_{tail}(\overline{t_n},k)$ is well-typed, we have that k has type C. Therefore, $f_{tail}(\overline{s_n},\mathsf{cont}_1(k,\overline{y_m}))$ is also well-typed when $C[f(\overline{s_n}]$ is well-typed in the original TRS.
  - Finally, let us consider a rule of the form $\mathsf{eval}(\mathsf{cont}_1(k,\overline{y_m}),w) \to \mathsf{eval}(k,C[w])$. By assuming that $\mathsf{eval}(\mathsf{cont}_1(k,\overline{y_m}),w)$ is well typed, we have that k has type C, $\overline{y_m}$ has type $\overline{\mathsf{D}_m}$, and w has type T. Therefore, $C[w]$ has type T when $f(\overline{t_n}) \to C[f(\overline{s_n})]$ is well-typed and, thus, $\mathsf{eval}(k,C[w])$ is well-typed too.

4. Discussion and Future Work

In this paper, we have presented a technique for transforming recursive functions—defined by a term rewriting system—into tail recursive form. We have proved the correctness of the transformation for constructor-based reduction (a form of innermost reduction) over constructor rewrite systems.

Despite the fact that converting a function to tail recursive form is a fundamental problem, the only well-known automatic transformation we are aware of in the literature is the CPS-transformation that introduces higher-order lambda abstractions. In this work, in contrast, we have presented a direct transformation that keeps the transformed program in the (first-order) term rewriting setting. As mentioned in Section 1, a similar approach can be found in Wand's seminal paper [5], where continuations are represented by data structures. However, the examples presented by Wand required some *eureka* steps and, therefore, no automatic technique is introduced. Of course, one could use a conventional CPS-transformation and, then, apply a *defunctionalization* post-process to remove higher-order expressions, as proposed by Danvy and Nielsen [9].[5] However, in this case, proving the correctness of the combined transformation would be much more involved than in our direct approach, since one should be able to deal with the intermediate higher-order term rewriting systems. Actually, to the best of our knowledge, this is the first fully automatic transformation that directly converts functions to tail recursive form in a (first-order) term rewriting setting.

An interesting topic for future work is the extension of the conversion algorithm in order to also introduce accumulators. Let us illustrate the aim of this extension with an example. Here, we consider an example from [5, 2] to generate the well-known reverse with an accumulating parameter from the naive reverse. We start with the following definition of function reverse:

$$\begin{array}{rcl}
\mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} \\
\mathsf{reverse}(x : xs) & \to & \mathsf{app}(\mathsf{reverse}(xs), [x]) \\
\mathsf{app}(\mathsf{nil}, ys) & \to & ys \\
\mathsf{app}(x : xs, ys) & \to & x : \mathsf{app}(xs, ys)
\end{array}$$

Transforming function reverse to tail recursion using function $\mathcal{T}ail$, we get

$$\begin{array}{rcl}
\mathsf{reverse}(xs) & \to & \mathsf{rev}(xs, \mathsf{id}) \\
\mathsf{rev}(\mathsf{nil}, k) & \to & \mathsf{eval}(k, \mathsf{nil}) \\
\mathsf{rev}(x : xs, k) & \to & \mathsf{rev}(xs, \mathsf{cont}(k, x)) \\
\mathsf{eval}(\mathsf{id}, ws) & \to & ws \\
\mathsf{eval}(\mathsf{cont}(k, x), ws) & \to & \mathsf{eval}(k, \mathsf{app}(ws, [x]))
\end{array}$$

[5]Nevertheless, it should be noted that the restriction to *constructor-based* reduction that we found essential for proving the correctness of the transformation has no counterpart in [9]. Therefore, the application of the general scheme in [9] to rewrite systems so that the original semantics is preserved is still not trivial.

together with the original definition of function app. Now, the key observation is that the expression $\mathsf{eval}(k, w)$ always reduces to $\mathsf{app}(w, ys)$ for some list ys (that depends on k). We can prove this inductively:

- The base case is trivial with list $ys = \mathsf{nil}$ since $\mathsf{eval}(\mathsf{id}, w) = w = \mathsf{app}(w, \mathsf{nil})$.
- For the inductive case, we assume that $\mathsf{eval}(k, w) = \mathsf{app}(w, ys)$ for some list ys. Now, we prove that the claim also holds for $\mathsf{eval}(\mathsf{cont}(k', x), w)$. By unfolding the call, we have:

  $$\mathsf{eval}(\mathsf{cont}(k', x), w) = \mathsf{eval}(k', \mathsf{app}(w, [x]))$$

  By applying the inductive hypothesis, we have

  $$\mathsf{eval}(k', \mathsf{app}(w, [x])) = \mathsf{app}(\mathsf{app}(w, [x]), ys)$$

  for some list ys. Finally, by the associativity of app and the definition of app, we have

  $$\mathsf{app}(\mathsf{app}(w, [x]), ys) = \mathsf{app}(w, \mathsf{app}([x], ys)) = \mathsf{app}(w, x : ys)$$

  and the claim follows.

Therefore, we do not really need to use the function eval and can safely replace it by the list ys above:

$$\begin{array}{rcl} \mathsf{reverse}(xs) & \to & \mathsf{rev}(xs, \mathsf{nil}) \\ \mathsf{rev}(\mathsf{nil}, ys) & \to & ys \\ \mathsf{rev}(x : xs, ys) & \to & \mathsf{rev}(xs, x : ys) \end{array}$$

Automating this process is clearly far from trivial and may require partial evaluation techniques for rewrite systems [10, 11, 12] (e.g., for finding *patterns* for continuation terms) as well as the assistance of a theorem prover (for proving that the proposed patterns are indeed true).

We consider the (semi-)automated introduction of accumulators a challeging topic for future work.

References

[1] N. Nishida, G. Vidal, Program inversion for tail recursive functions, in: M. Schmidt-Schauß (Ed.), Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA 2011), Vol. 10 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011, pp. 283–298.

[2] A. Field, P. Harrison, Functional Programming, Addison-Wesley, 1988.

[3] G. D. Plotkin, Call-by-name, call-by-value and the lambda-calculus, Theor. Comput. Sci. 1 (2) (1975) 125–159.

[4] G. L. Steele, Rabbit: A compiler for Scheme (M. Sc. Thesis), Tech. Rep. AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts (May 1978).

[5] M. Wand, Continuation-based program transformation strategies, J. ACM 27 (1) (1980) 164–180.

[6] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.

[7] E. Ohlebusch, Advanced topics in term rewriting, Springer, 2002.

[8] P. Schneider-Kamp, J. Giesl, A. Serebrenik, R. Thiemann, Automated termination proofs for logic programs by term rewriting, ACM Trans. Comput. Log. 11 (1) (2009) 1–52.

[9] O. Danvy, L. R. Nielsen, Defunctionalization at work, in: PPDP, ACM, 2001, pp. 162–174.

[10] E. Albert, G. Vidal, The narrowing-driven approach to functional logic program specialization, New Generation Computing 20 (1) (2002) 3–26.

[11] A. Bondorf, Compiling Laziness by Partial Evaluation, in: S. P. Jones, G. Hutton, C. K. Holst (Eds.), Functional Programming, Glasgow 1990, Springer-Verlag, Berlin, 1991, pp. 9–22.

[12] J. G. Ramos, J. Silva, G. Vidal, Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems, in: O. Danvy, B. C. Pierce

(Eds.), Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming (ICFP'05), ACM Press, 2005, pp. 228–239.