Abstract

We identify the problem of automatic recursion removal using the unfold/fold methodology (2) to the problem of finding a ε-pattern-matcher σ for two terms t and s, i.e. so that σt = s. We propose a new method to solve this equation based on a technique of dynamic completion of a term rewriting system for the property t = s ⇒ t ⪰ Rs. This method presents some advantages because it enables us to work with incomplete theories and limits the number of superpositions we must do during the process of completion.

1. Introduction

Program transformations are now widely accepted as a powerful tool to design correct and efficient programs (1, 2, 4). The idea is to start with a simple and correct program ignoring any question of efficiency and then try to improve it using correctness preserving transformations. This methodology originated some systems which, interactively, help programmers in developing their programs. One of the best known is due to Burstall and Darlington (2). It is based on the heuristic use of few and simple transformation rules and can do many optimizing transformations such as combining loops re-use storage, recursion removal or even synthesising implementations for abstract data types (4). In this paper we consider only the problem of recursion removal. Our aim is to show how we can fully automatize this transformation for the class of linear recursive definitions. For this purpose we have developed a new technique for using term rewriting systems which, in our case, is more efficient than the traditional Knuth-Bendix (9) technique.

2. Recursion removal using Burstall-Darlington methodology

In Burstall-Darlington system recursion removal is based on the use of a sequence of three transformation rules, named unfold, laws and fold, and defined as follows:

(i) unfold: if E ⇔ E' and F ⇔ F' are equations and there is some occurrence in F' of an instance of E, replace it by the corresponding instance of E' obtaining F'.
then add the equation \( F \Leftrightarrow F^* \);

(ii) laws: we may transform any equation by using on its right hand expression any
we have about the base functions (associativity, commutativity, etc.);

(iii) fold: it is the reverse of the unfold transformation.

Removing recursion relies on the user's ability to invent a new (auxiliary) function — called "eureka" by Burstall-Darlington — and on his intuition for finding a
tail-recursive definition for it.

Example 1

Consider the following linear recursive definition \((x, n \in \mathbb{N} \text{ and } n \geq 0)\):

\[
F(x, n) = \begin{cases} 
1 & \text{if } n=0 \\
F(x, n+2)^2 & \text{if } \text{even}(n) \\
(F(x, (n-1)+2)^2)*x & \text{else}
\end{cases} 
\]  

\tag{1}

In order to remove this non tail-recursion the user must provide the system with
the auxiliary function:

\[
G(g_1, g_2, g_3, g_4) = (F(g_1, g_2)^g_3)*g_4 
\]  

\tag{2}

which can be used to compute \(F(x, n)\) because

\[
F(x, n) = G(x, n, 1, 1) 
\]  

\tag{3}

The next problem to solve is to find a tail-recursive definition for \(G\). This is a-
chieved by the system (under user's guidance) by unfolding once the \(F\) call in (2), u-
sing (1) and rewriting the unfolded expression in such a way that finally it can be
folded using (2):

\[
G(g_1, g_2, g_3, g_4) = \begin{cases} 
1 & \text{if } g_2=0 \\
G(g_1, g_2/2)^2 & \text{if } \text{even}(g_2) \\
(G(g_1, (g_2-1)+2)^2)*g_1 & \text{else}
\end{cases} 
\]  

by unfolding \(F\) in (2).

Assuming that the if-then-else-fi is naturally extended (10) this can be rewritten as:

\[
= \begin{cases} 
(1^g_3)*g_4 & \text{if } g_2=0 \\
((F(g_1, g_2/2)^2)^g_3)*g_4 & \text{if } \text{even}(g_2) \\
(((F(g_1, (g_2-1)+2)^2)*g_1)^g_3)*g_4 & \text{else}
\end{cases} 
\]  

\tag{2'}

38