Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Dependency pairs for proving termination properties of conditional term rewriting systems ☆

Salvador Lucas [a,*], José Meseguer [b]

[a] DSIC, Universitat Politècnica de València, Spain
[b] CS Dept. at the University of Illinois at Urbana-Champaign, United States

A R T I C L E   I N F O

Article history:
Received 5 December 2014
Received in revised form 23 March 2016
Accepted 24 March 2016
Available online xxxx

Dedicated to the memory of Bernhard Gramlich

Keywords:
Conditional term rewriting
Dependency pairs
Program analysis
Operational termination

A B S T R A C T

The notion of *operational termination* provides a logic-based definition of termination of computational systems as the absence of *infinite inferences* in the computational logic describing the *operational semantics* of the system. For *Conditional Term Rewriting Systems* we show that operational termination is characterized as the conjunction of *two* termination properties. One of them is traditionally called *termination* and corresponds to the absence of infinite sequences of rewriting steps (a *horizontal* dimension). The other property, that we call V*-termination*, concerns the absence of infinitely many attempts to launch the *subsidiary processes* that are required to perform a *single* rewriting step (a *vertical* dimension). We introduce appropriate notions of *dependency pairs* to characterize termination, V-termination, and operational termination of Conditional Term Rewriting Systems. This can be used to obtain a powerful and more expressive framework for proving termination properties of Conditional Term Rewriting Systems.

1. Introduction

Conditional Term Rewriting Systems (CTRSs [6,11,24]) extend Term Rewriting Systems (TRSs [5,36,41]) by adding a (possibly empty) *conditional* part c to each rewrite rule $\ell \to r$, thus obtaining a *conditional rewrite rule* $\ell \to r \Leftarrow c$. The addition of such conditional parts c substantially increases the expressiveness of programming languages that use them (e.g., ASF+SDF [8], CafeOBJ [15], ELAN [7], Haskell [23], OBJ [19], or Maude [9]) and often clarifies the purpose of the rules to make programs more readable and self-explanatory. For instance, in functional programs, the use of *guards* and *local definitions* (by means of `where` clauses) is customary.

Example 1. The following Haskell program implements the well-known *quicksort* algorithm [36, Section 1]:

```
split x [] = ([],[])
split x (y:ys)
   | x <= y = (xs,y:zs)
   | otherwise = (y:xs,zs)
   where (xs,zs) = split x ys
```

☆ Partially supported by the EU (FEDER), Spanish MINECO projects TIN 2013-45732-C4-1-P and TIN2015-69175-C4-1-R, GV project PROMETEOII/2015/013, and NSF grant CNS 13-19109. Salvador Lucas' research was partly developed during a sabbatical year at UIUC.
* Corresponding author.
E-mail address: slucas@dsic.upv.es (S. Lucas).

http://dx.doi.org/10.1016/j.jlamp.2016.03.003
2352-2208/

```
qsort [] = []
qsort (x:xs) = qsort ys ++ (x:qsort zs)
    where (ys,zs) = split x xs
```

This program can be understood as a CTRS (borrowing from [36, Section 1]; we have added rules to compare natural numbers in Peano's notation (with leq), and for implementing Haskell's *appending* operator ++ for lists with app):

$$\mathsf{leq}(0,x) \to \mathsf{true} \tag{1}$$

$$\mathsf{leq}(\mathsf{s}(x),0) \to \mathsf{false} \tag{2}$$

$$\mathsf{leq}(s(x),\mathsf{s}(y)) \to \mathsf{leq}(x,y) \tag{3}$$

$$\mathsf{app}(\mathsf{nil},xs) \to xs \tag{4}$$

$$\mathsf{app}(\mathsf{cons}(x,xs),ys) \to \mathsf{cons}(x,\mathsf{app}(xs,ys)) \tag{5}$$

$$\mathsf{split}(x,\mathsf{nil}) \to \mathsf{pair}(\mathsf{nil},\mathsf{nil}) \tag{6}$$

$$\begin{aligned}\mathsf{split}(x,\mathsf{cons}(y,ys)) &\to \mathsf{pair}(xs,\mathsf{cons}(y,zs))\\ &\Leftarrow \mathsf{leq}(x,y) \to \mathsf{true}, \mathsf{split}(x,ys) \to \mathsf{pair}(xs,zs)\end{aligned} \tag{7}$$

$$\begin{aligned}\mathsf{split}(x,\mathsf{cons}(y,ys)) &\to \mathsf{pair}(\mathsf{cons}(y,xs),zs)\\ &\Leftarrow \mathsf{leq}(x,y) \to \mathsf{false}, \mathsf{split}(x,ys) \to \mathsf{pair}(xs,zs)\end{aligned} \tag{8}$$

$$\mathsf{qsort}(\mathsf{nil}) \to \mathsf{nil} \tag{9}$$

$$\begin{aligned}\mathsf{qsort}(\mathsf{cons}(x,xs)) &\to \mathsf{app}(\mathsf{qsort}(ys),\mathsf{cons}(x,\mathsf{qsort}(zs)))\\ &\Leftarrow \mathsf{split}(x,xs) \to \mathsf{pair}(ys,zs)\end{aligned} \tag{10}$$

Note the following:

1. a guard b in the Haskell program (e.g., `x <= y` and `otherwise`, which here means that the condition `x <= y` does *not* hold) is translated as a boolean test $b \to^* \mathsf{true}$ or $b \to^* \mathsf{false}$. The intended meaning is that the boolean expression b is evaluated by rewriting (in zero or more steps, denoted as $\to^*$) and then the outcome is checked to see whether it is true or false, respectively.
2. `where` clauses defining *pattern matching conditions* $p = e$ for an expression e whose *value* is matched against a pattern p are translated as rewriting conditions $e \to^* p$. The intended meaning is that e will be evaluated and the outcome matched against p. In this way, variables in p become instantiated to expressions which are then used in the right-hand side of the rule to return the final result of the computation. Thus, part of such a computation is accomplished in the conditional part of the rules.

The example illustrates two practical uses of conditional rules when defining functions:

1. Testing *boolean conditions* before applying a rule, as in (7) and (8).
2. Local *reductions* of specific expressions followed by *matching* against a pattern in order to obtain pieces of information which can be used to build the outcome as in rules (7), (8), and (10).

Although several *transformations* have been envisaged to *remove* the conditional part of the rules, thus yielding an 'equivalent' TRS (see [31,35,37,39] and the references therein), programmers still find conditional rules valuable when writing programs in the aforementioned languages.

1.1. Termination, V-termination, and operational termination of CTRSs

The semantics of rewriting-based computational systems is often described by means of the transitions induced by the *rewriting steps*. The *one-step rewriting relation* $\to_{\mathcal{R}}$ on terms induced by a CTRS $\mathcal{R}$ is the basis to describe any accomplished *evaluation* or *transformation* of expressions. In this setting, the *absence of infinite rewrite sequences* $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$ arises as a natural definition of *terminating behavior* for CTRSs. However, computations with CTRSs with rules $\ell \to r \Leftarrow s_1 \to t_1, \ldots, s_n \to t_n$ (i.e., the conditional part of a rule consists of a sequence of pairs $s_i \to t_i$, for $1 \le i \le n$) are defined by means