Applying Rewriting Techniques to the Verification of Erlang Processes[*]

Thomas Arts[1] and Jürgen Giesl[2]

[1] Computer Science Laboratory, Ericsson Utvecklings AB, Box 1505, 125 25 Älvsjö,
Sweden, E-mail: `thomas@cslab.ericsson.se`
[2] Dept. of Computer Science, Darmstadt University of Technology, Alexanderstr. 10,
64283 Darmstadt, Germany, E-mail: `giesl@informatik.tu-darmstadt.de`

Abstract. Erlang is a functional programming language developed by
Ericsson Telecom which is particularly well suited for implementing con-
current processes. In this paper we show how methods from the area
of term rewriting are presently used at Ericsson. To verify properties of
processes, such a property is transformed into a termination problem of
a conditional term rewriting system (CTRS). Subsequently, this termi-
nation proof can be performed automatically using *dependency pairs*.
The paper illustrates how the dependency pair technique can be applied
for termination proofs of *conditional* TRSs. Secondly, we present two
refinements of this technique, viz. *narrowing* and *rewriting dependency
pairs*. These refinements are not only of use in the industrial application
sketched in this paper, but they are generally applicable to arbitrary
(C)TRSs. Thus, in this way dependency pairs can be used to prove ter-
mination of even more (C)TRSs automatically.

1 Introduction

In a patent application [HN99], Ericsson developed a new protocol for distributed
telecommunication processes. This paper originates from an attempt to verify
this protocol's implementation written in Erlang. To save resources and to in-
crease reliability, the aim was to perform as much as possible of this verification
automatically. Model checking techniques were not applicable, since the property
to be proved requires the consideration of the infinite state space of the process. A
user guided approach based on theorem proving was successful, but very labour
intensive [AD99]. We describe one of the properties which had to be verified in
Sect. 2 and show that it can be represented as a non-trivial termination problem
of a CTRS. But standard techniques (see e.g. [Der87,Ste95,DH95]) and even re-
cent advances like the dependency pair technique [AG97a,AG97b,AG98,AG99]
could not perform the required termination proof automatically.

In Sect. 3 we show that termination problems of CTRSs can be reduced to
termination problems of unconditional TRSs. After recapitulating the basic no-
tions of dependency pairs in Sect. 4, we present two important extensions, viz.
narrowing (Sect. 5) and *rewriting* dependency pairs (Sect. 6) which are partic-
ularly useful in the context of CTRSs. With these refinements, the dependency
pair approach could solve the process verification problem automatically.

---

2 A Process Verification Problem

We have to prove properties of a process in a network. The process receives messages which consist of a list of data items and an integer M. For every item in the list, the process computes a new list of data items. For example, the data items could be telephone numbers and the process could generate a list of calls to that number on a certain date. The resulting list may have arbitrary length, including zero. The integer M in the message indicates how many items of the newly computed list should be sent to the next process. The restriction on the number of items that may be sent out is imposed for practical optimization reasons.

Of course, the process may have computed more than M new items and in that case, it stores the remaining answers in an accumulator (implemented by an extra argument Store of the process). However, whenever it has sent the first M items to the next process, our process may receive a new message. To respond to the new message, the process first checks whether its store already contains at least M items. In this case, it sends the first M items from its store and depending on the incoming message, probably some new items are computed afterwards. Otherwise, if the store contains fewer than M items, then the next process has to wait until the new items are computed. After this computation, the first M items from the newly obtained item list and the store are sent on to the next process. Again, those items that our process could not send out are stored in its accumulator.

Finally, in order to empty the store, the empty list is sent to our process repeatedly. In the end, so is the claim, this process will send the empty list as well. This article describes how we are able to formally and automatically verify this claim. The Erlang code is given below (because of space limitations the code for obvious library functions like append and leq is not presented).

```
process(NextPid,Store) ->
 receive {Items,M} ->
  case leq(M,length(Store)) of
  true -> {ToSend,ToStore} = split(M,Store),
          NextPid!{ToSend,M},
          process(NextPid,append(map_f(self(),Items),ToStore));
  false ->{ToSend,ToStore} = split(M,append(map_f(self(),Items),Store)),
          NextPid!{ToSend,M},
          process(NextPid,ToStore)
  end
 end.

map_f(Pid,nil) -> nil;
map_f(Pid,cons(H,T)) -> append(f(Pid,H),map_f(Pid,T)).
```

For a list L, split(M,L) returns a pair of lists $\{L_1, L_2\}$ where L_1 contains the first M elements (or L if its length is shorter than M) and L_2 contains the rest of L. The command '!' denotes the sending of data and NextPid!{ToSend,M} stands for sending the items ToSend and the integer M to the process with the identifier NextPid. A process can obtain its own identifier by calling the function self(). For every item in the list Items, the function map_f(Pid,Items) computes new

data items by means of the function `f(Pid,Item)`. So the actual computation that `f` performs depends on the process identifier `Pid`. Hence, to compute new data items for the incoming `Items`, our process has to pass its own identifier to the function `map_f`, i.e., it calls `map_f(self(),Items)`.

Note that this process itself is not a terminating function: in fact, it has been designed to be non-terminating. Our aim is not to prove its termination, but to verify a certain property, which can be expressed in terms of termination. As part of the correctness proof of the software, we have to prove that if the process continuously receives the message `{nil,M}` for any integer `M`, then eventually the process will send the message `{nil,M}` as well. This property must hold independent of the value of the store and of the way in which new data items are generated from given ones. Therefore, `f` has been left unspecified, i.e., `f` may be any terminating function which returns a list of arbitrary length.

The framework of term rewriting [DJ90,BN98] is very useful for this verification. We prove the desired property by constructing a CTRS containing a binary function `process` whose arguments represent the stored data items `Store` and the integer `M` sent in the messages. In this example, we may abstract from the process communication. Thus, the Erlang function `self()` becomes a constant and we drop the send command (`!`) and the argument `NextPid` in the CTRS. Since we assume that the process constantly receives the message `{nil,M}`, we hard-code it into the CTRS. Thus, the variable `Items` is replaced by nil. As we still want to reason about the variable `M`, we added it to the arguments of the `process`. To model the function `split` (which returns a *pair* of lists) in the CTRS, we use separate functions fstsplit and sndsplit for the two components of `split`'s result. Now the idea is to force the function `process` to terminate if `ToSend` is the empty list nil. So we only continue the computation if application of the function empty to the result of fstsplit yields false. Thus, if all evaluations w.r.t. this CTRS terminate, then the original process eventually outputs the demanded value.

$$\mathsf{leq}(m, \mathsf{length}(store)) \to^* \mathsf{true}, \quad \mathsf{empty}(\mathsf{fstsplit}(m, store)) \to^* \mathsf{false} \quad |$$
$$\mathsf{process}(store, m) \to \mathsf{process}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(m, store)), m) \quad (1)$$

$$\mathsf{leq}(m, \mathsf{length}(store)) \to^* \mathsf{false}, \quad \mathsf{empty}(\mathsf{fstsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store))) \to^* \mathsf{false} \quad |$$
$$\mathsf{process}(store, m) \to \mathsf{process}(\mathsf{sndsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store)), m) \quad (2)$$

The auxiliary Erlang functions as well as the functions for empty, fstsplit, and sndsplit are straightforwardly expressed by unconditional rewrite rules.

$$\begin{aligned}
\mathsf{length}(\mathsf{nil}) &\to 0 & \mathsf{sndsplit}(0, x) &\to x \\
\mathsf{length}(\mathsf{cons}(h, t)) &\to \mathsf{s}(\mathsf{length}(t)) & \mathsf{sndsplit}(\mathsf{s}(n), \mathsf{nil}) &\to \mathsf{nil} \\
\mathsf{fstsplit}(0, x) &\to \mathsf{nil} & \mathsf{sndsplit}(\mathsf{s}(n), \mathsf{cons}(h, t)) &\to \mathsf{sndsplit}(n, t) \\
\mathsf{fstsplit}(\mathsf{s}(n), \mathsf{nil}) &\to \mathsf{nil} & \mathsf{empty}(\mathsf{nil}) &\to \mathsf{true} \\
\mathsf{fstsplit}(\mathsf{s}(n), \mathsf{cons}(h, t)) &\to \mathsf{cons}(h, \mathsf{fstsplit}(n, t)) & \mathsf{empty}(\mathsf{cons}(h, t)) &\to \mathsf{false} \\
\mathsf{app}(\mathsf{nil}, x) &\to x & \mathsf{leq}(0, m) &\to \mathsf{true} \\
\mathsf{app}(\mathsf{cons}(h, t), x) &\to \mathsf{cons}(h, \mathsf{app}(t, x)) & \mathsf{leq}(\mathsf{s}(n), 0) &\to \mathsf{false} \\
\mathsf{map\_f}(pid, \mathsf{nil}) &\to \mathsf{nil} & \mathsf{leq}(\mathsf{s}(n), \mathsf{s}(m)) &\to \mathsf{leq}(n, m) \\
\mathsf{map\_f}(pid, \mathsf{cons}(h, t)) &\to \mathsf{app}(\mathsf{f}(pid, h), \mathsf{map\_f}(pid, t))
\end{aligned}$$

The rules for the Erlang function f are not specified, since we have to verify the desired property for *any* terminating function f. However, as Erlang has an eager (call-by-value) evaluation strategy, if a terminating Erlang function f is straightforwardly transformed into a (C)TRS (such as the above library functions), then any evaluation w.r.t. these rules is finite. Now to prove the desired property of the Erlang process, we have to show that the whole CTRS with all its extra rules for the auxiliary functions only permits finite evaluations.

The construction of the above CTRS is rather straightforward, but it presupposes an understanding of the program and the verification problem and therefore it can hardly be mechanized. But after obtaining the CTRS, the proof that any evaluation w.r.t. this CTRS is finite should be done automatically.

In this paper we describe an extension of the dependency pair technique which can perform such automatic proofs. Moreover, this extension is of general use for termination proofs of TRSs and CTRSs. Hence, our results significantly increase the class of systems where termination can be shown mechanically.

3 Termination of Conditional Term Rewriting Systems

A CTRS is a TRS where conditions $s_1 = t_1, \ldots, s_n = t_n$ may be added to rewrite rules $l \to r$. In this paper, we restrict ourselves to CTRSs where all variables in the conditions s_i, t_i also occur in l. Depending on the interpretation of the equality sign in the conditions, different rewrite relations can be associated with a CTRS, cf. e.g. [Kap84,BK86,DOS88,BG89,DO90,Mid93,Gra94,SMI95,Gra96a, Gra96b]. In our verification example, we transformed the problem into an *oriented* CTRS [SMI95], where the equality signs in conditions of rewrite rules are interpreted as reachability ($\to^*$). Thus, we denote rewrite rules by

$$s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r. \tag{3}$$

In fact, we even have a *normal* CTRS, because all t_i are ground normal forms w.r.t. the TRS which results from dropping all conditions.

A reduction of $C[l\sigma]$ to $C[r\sigma]$ with rule (3) is only possible if $s_i\sigma$ reduces to $t_i\sigma$ for all $1 \leq i \leq n$. Formally, the rewrite relation $\to_{\mathcal{R}}$ of a CTRS $\mathcal{R}$ can be defined as $\to_{\mathcal{R}} = \bigcup_{j \geq 0} \to_{\mathcal{R}_j}$, where $\mathcal{R}_0 = \emptyset$ and $\mathcal{R}_{j+1} = \{l\sigma \to r\sigma \mid s_i\sigma \to^*_{\mathcal{R}_j} t_i\sigma$ for all $1 \leq i \leq n$ and some rule (3) in $\mathcal{R}\}$, cf. e.g. [Mid93,Gra96b].

A CTRS $\mathcal{R}$ is *terminating* iff $\to_{\mathcal{R}}$ is well founded. But termination is not enough to ensure that every evaluation with a CTRS is finite. For example, assume that evaluation of the condition leq(m, length($store$)) in our CTRS would require the reduction of process($store, m$). Then evaluation of process($store, m$) would yield an infinite computation. Nevertheless, process($store, m$) could not be rewritten further and thus, the CTRS would be terminating. But in this case, the desired property would *not* hold for the original Erlang process, because this would correspond to a deadlock situation where no messages are sent out at all.

For that reason, instead of *termination* one is often much more interested in *decreasing* CTRSs [DOS88]. In this paper, we use a slightly modified notion of

4

decreasingness, because in our evaluation strategy conditions are checked from left to right, cf. [WG94]. Thus, the i-th condition $s_i \to^* t_i$ is only checked if all previous conditions $s_j \to^* t_j$ for $1 \le j < i$ hold.

Definition 1 (Left-Right Decreasing). *A CTRS $\mathcal{R}$ is left-right decreasing if there exists a well-founded relation $>$ containing the rewrite relation $\to_{\mathcal{R}}$ and the subterm relation $\rhd$ such that $l\sigma > s_i\sigma$ holds for all rules like (3), all $i \in \{1, \ldots, n\}$, and all substitutions σ where $s_j\sigma \to_{\mathcal{R}}^* t_j\sigma$ for all $j \in \{1, \ldots, i-1\}$.*

This definition of left-right decreasingness exactly captures the finiteness of recursive evaluation of terms. (Obviously, decreasingness implies left-right decreasingness, but not vice versa.) Hence, now our aim is to prove that the CTRS corresponding to the Erlang process is left-right decreasing.

A standard approach for proving termination of a CTRS $\mathcal{R}$ is to verify termination of the TRS $\mathcal{R}'$ which results from dropping all conditions (and for decreasingness one has to impose some additional demands). But this approach fails for CTRSs where the conditions are necessary to ensure termination. This also happens in our example, because without the conditions $\mathsf{empty}(\ldots) \to^* \mathsf{false}$ the CTRS is no longer terminating (and thus, not left-right decreasing either).

A solution for this problem is to transform CTRSs into *unconditional* TRSs, cf. [DP87,GM87,Mar96]. For unconditional rules, let $\mathrm{tr}(l \to r) = \{l \to r\}$. If ϕ is a conditional rule, i.e., $\phi = \text{'}s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r\text{'}$, we define $\mathrm{tr}(\phi) =$

$$\{l \to \mathsf{if}_{1,\phi}(\boldsymbol{x}, s_1)\} \cup \{\mathsf{if}_{i,\phi}(\boldsymbol{x}, t_i) \to \mathsf{if}_{i+1,\phi}(\boldsymbol{x}, s_{i+1}) \mid 1 \le i < n\} \cup \{\mathsf{if}_{n,\phi}(\boldsymbol{x}, t_n) \to r\},$$

where $\boldsymbol{x}$ is the tuple of all variables in l and the if's are new function symbols. To ease readability we often just write if_n for some $n \in \mathbb{N}$ where if_n is a function symbol which has not been used before.

Let $\mathcal{R}^{\mathrm{tr}} = \bigcup_{\phi \in \mathcal{R}} \mathrm{tr}(\phi)$. For CTRSs without extra variables, $\mathcal{R}^{\mathrm{tr}}$ is indeed an (unconditional) TRS. (An extension to *deterministic* CTRSs [BG89] with extra variables is also possible.) The transformation of Rule (1) results in

$$\mathsf{process}(store, m) \to \mathsf{if}_1(store, m, \mathsf{leq}(m, \mathsf{length}(store))) \tag{4}$$

$$\mathsf{if}_1(store, m, \mathsf{true}) \to \mathsf{if}_2(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, store))) \tag{5}$$

$$\mathsf{if}_2(store, m, \mathsf{false}) \to \mathsf{process}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(m, store)), m). \tag{6}$$

Now we aim to prove termination of $\mathcal{R}^{\mathrm{tr}}$ instead of $\mathcal{R}$'s left-right decreasingness.

In [GM87], this transformation is restricted to a limited class of convergent CTRSs. However, in the following we show that for our purpose this restriction is not necessary. In other words, termination of $\mathcal{R}^{\mathrm{tr}}$ indeed implies left-right decreasingness (and thus also termination) of $\mathcal{R}$. Thus, this transformation is a generally applicable technique to reduce the termination problem of CTRSs to a termination problem of unconditional TRSs. (A similar approach was presented in [Mar96] for decreasingness proofs (instead of *left-right* decreasingness) by using a transformation where all conditions of a rule have to be checked in parallel.) We first prove that any reduction with $\mathcal{R}$ can be simulated by $\mathcal{R}^{\mathrm{tr}}$.

Lemma 1. *Let q, q' be terms without if's. If $q \to_{\mathcal{R}}^+ q'$, then $q \to_{\mathcal{R}^{\mathrm{tr}}}^+ q'$.*

Proof. There must be a $j \in \mathbb{N}$ such that $q \to_{\mathcal{R}_j}^+ q'$ (j is the *depth* of the reduction). We prove the theorem by induction on the depth and the length of the reduction $q \to_{\mathcal{R}}^+ q'$ (i.e., we use a lexicographic induction relation).

The reduction has the form $q \to_{\mathcal{R}} p \to_{\mathcal{R}}^* q'$ and by the induction hypothesis we know $p \to_{\mathcal{R}^{\mathrm{tr}}}^* q'$. Thus, it suffices to prove $q \to_{\mathcal{R}^{\mathrm{tr}}}^+ p$.

If the reduction $q \to_{\mathcal{R}} p$ is done with an unconditional rule of $\mathcal{R}$, then the conjecture is trivial. Otherwise, we must have $q = C[l\sigma]$, $p = C[r\sigma]$ for some context C and some rule like (3). As the depth of the reductions $s_i\sigma \to_{\mathcal{R}}^* t_i\sigma$ is less than the depth of the reduction $q \to_{\mathcal{R}}^+ q'$, by the induction hypothesis we have $s_i\sigma \to_{\mathcal{R}^{\mathrm{tr}}}^* t_i\sigma$. This implies $q \to_{\mathcal{R}^{\mathrm{tr}}}^+ p$. □

Now the desired result is a direct consequence of Lemma 1.

Corollary 1 (Left-Right Decreasing of $\mathcal{R}$ by Termination of $\mathcal{R}^{\mathrm{tr}}$). *If $\mathcal{R}^{\mathrm{tr}}$ is terminating, then $\mathcal{R}$ is left-right decreasing (and thus, it is also terminating).*

Proof. If $\to_{\mathcal{R}^{\mathrm{tr}}}$ is well founded, then $\to_{\mathcal{R}^{\mathrm{tr}}} \cup \rhd$ and hence, the transitive closure $(\to_{\mathcal{R}^{\mathrm{tr}}} \cup \rhd)^+$ are well founded, too. By Lemma 1, this relation satisfies all conditions imposed on the relation $>$ in Def. 1. Hence, $\mathcal{R}$ is left-right decreasing. □

In our example, the conditional rule (2) is transformed into three additional unconditional rules. But apart from the if-root symbol of the right-hand side, the first of these rules is identical to (4). Thus, we obtain two overlapping rules in the transformed TRS which correspond to the overlapping conditional rules (1) and (2). However, in the CTRS this critical pair is *infeasible* [DOS88], i.e., the conditions of both rules exclude each other. Thus, our transformation of CTRSs into TRSs sometimes introduces unnecessary rules and overlap.

Therefore, whenever we construct a rule of the form $q \to \mathsf{if}_k(t)$ and there already exists a rule $q \to \mathsf{if}_n(t)$, then we identify if_k and if_n. This does not affect the soundness of our approach, because termination of a TRS where all occurrences of a symbol g are substituted by a symbol f with the same arity always implies termination of the original TRS.[1] Thus, we obtain the additional rules:

$$\mathsf{if}_1(store, m, \mathsf{false}) \to \mathsf{if}_3(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store)))) \quad (7)$$

$$\mathsf{if}_3(store, m, \mathsf{false}) \to \mathsf{process}(\mathsf{sndsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store)), m) \quad (8)$$

If termination of a CTRS depends on its conditions, then in general termination of the transformed TRS can only be shown if one examines which terms may follow each other in a reduction. However, in the classical approaches based on simplification orderings (cf. e.g. [Der87,Ste95]), such considerations do not take place. Hence, they fail in proving the termination of (4)-(8). For this reason, such transformations into unconditional TRSs have rarely been applied for

---

[1] This possibility to eliminate unnecessary overlap is an advantage of our transformation compared to the one of [Mar96], where the transformed unconditional TRSs remain overlapping. In practice, proving termination of non-overlapping TRSs is significantly easier, since one may use techniques specifically tailored to *innermost* termination proofs, see below.

termination (or decreasingness) proofs of CTRSs. However, we will demonstrate that with the *dependency pair* approach this transformation is very useful.

To verify our original goal, we now have to prove termination of the transformed TRS which consists of (4)-(8), the rules for all auxiliary (library) functions from Sect. 2, and the (unknown) rules for the unspecified function f. Note that if an Erlang function is straightforwardly transformed into a TRS, then this TRS is non-overlapping. Thus, we assume that all possible rules for the unspecified function f are non-overlapping as well. Then it is sufficient just to prove *innermost* termination of the resulting TRS, cf. e.g. [Gra95]. In order to apply verification on a large scale, the aim is to perform such proofs automatically. Extending the dependency pair technique makes this possible.

4 Dependency Pairs

Dependency pairs allow the use of existing techniques like simplification orderings for automated termination and innermost termination proofs where they were not applicable before. In this section we briefly recapitulate the basic concepts of this approach and we present the theorems that we need for the rest of the paper. For further details and explanations see [AG97b,AG98,AG99].

In contrast to the standard approaches for termination proofs, which compare left and right-hand sides of rules, we only examine those subterms that are responsible for starting new reductions. For that purpose we concentrate on the subterms in the right-hand sides of rules that have a defined[2] root symbol, because these are the only terms a rewrite rule can ever be applied to.

More precisely, for every rule $f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_m)]$ (where f and g are defined symbols), we compare the argument tuples $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$. To avoid the handling of tuples, for every defined symbol f we introduce a fresh *tuple* symbol F. To ease readability, we assume that the original signature consists of lower case function symbols only, whereas the tuple symbols are denoted by the corresponding upper case symbols. Now instead of the tuples $s_1, \ldots, s_n$ and $t_1, \ldots, t_m$ we compare the *terms* $F(s_1, \ldots, s_n)$ and $G(t_1, \ldots, t_m)$.

Definition 2 (Dependency Pair). *If* $f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_m)] \in \mathcal{R}$ *and g is defined, then* $\langle F(s_1, \ldots, s_n), G(t_1, \ldots, t_m) \rangle$ *is a* dependency pair *of* $\mathcal{R}$.

For the rules (4)-(8), (besides others) we obtain the following dependency pairs.

$$\langle \mathsf{PROCESS}(store, m), \mathsf{IF}_1(store, m, \mathsf{leq}(m, \mathsf{length}(store))) \rangle \tag{9}$$

$$\langle \mathsf{IF}_1(store, m, \mathsf{true}), \mathsf{IF}_2(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, store))) \rangle \tag{10}$$

$$\langle \mathsf{IF}_2(store, m, \mathsf{false}), \mathsf{PROCESS}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(m, store)), m) \rangle \tag{11}$$

$$\langle \mathsf{IF}_1(store, m, \mathsf{false}), \mathsf{IF}_3(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store)))) \rangle \tag{12}$$

$$\langle \mathsf{IF}_3(store, m, \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(m, \mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), store)), m) \rangle \tag{13}$$

To trace newly introduced redexes in an innermost reduction, we consider special sequences of dependency pairs, so-called *innermost chains*.

---

[2] Root symbols of left-hand sides are *defined* and all other functions are *constructors*.

Definition 3 (Innermost $\mathcal{R}$-chains). *Let $\mathcal{R}$ be a TRS. A sequence of dependency pairs $\langle s_1, t_1 \rangle \langle s_2, t_2 \rangle \ldots$ is called an* innermost $\mathcal{R}$-chain *if there exists a substitution σ, such that all $s_j\sigma$ are in normal form and $t_j\sigma \xrightarrow{i}^{*}_{\mathcal{R}} s_{j+1}\sigma$ holds for every two consecutive pairs $\langle s_j, t_j \rangle$ and $\langle s_{j+1}, t_{j+1} \rangle$ in the sequence.*

We always assume that different (occurrences of) dependency pairs have disjoint variables and we always regard substitutions whose domains may be infinite. In [AG97b] we showed that the absence of infinite innermost chains is a (sufficient and necessary) criterion for innermost termination. To improve this criterion we introduced the following graph which contains arcs between all those dependency pairs which may follow each other in innermost chains.

Definition 4 (Estimated Innermost Dependency Graph). *Let $\textsc{cap}(t)$ result from t by replacing all subterms with defined root symbols by different fresh variables. The* estimated innermost dependency graph *is the directed graph whose nodes are the dependency pairs and there is an arc from $\langle s, t \rangle$ to $\langle v, w \rangle$ iff $\textsc{cap}(t)$ and v are unifiable by a mgu μ where $s\mu$ and $v\mu$ are normal forms. A non-empty set $\mathcal{P}$ of dependency pairs is called a* cycle *iff for all $\langle s, t \rangle, \langle v, w \rangle \in \mathcal{P}$, there is a path from $\langle s, t \rangle$ to $\langle v, w \rangle$ in this graph, which only traverses pairs from $\mathcal{P}$.*

In our example, (besides others) there are arcs from (9) to (10) and (12), from (10) to (11), from (12) to (13), and from both (11) and (13) to (9). Thus, the dependency pairs (9)-(13) form the cycles $\mathcal{P}_1 = \{(9), (10), (11)\}$, $\mathcal{P}_2 = \{(9), (12), (13)\}$, and $\mathcal{P}_3 = \{(9), (10), (11), (12), (13)\}$. However, (9)-(13) are not on a cycle with any *other* dependency pair (e.g., dependency pairs from the rules of the auxiliary library functions or the unspecified function f, since we assume that f does not call process). This leads to the following refined criterion.

Theorem 1 (Innermost Termination Criterion). *A finite TRS $\mathcal{R}$ is innermost terminating iff for each cycle $\mathcal{P}$ in the estimated innermost dependency graph there exists no infinite innermost $\mathcal{R}$-chain of dependency pairs from $\mathcal{P}$.*

Note that in our definition, a cycle is a *set* of dependency pairs. Thus, for a finite TRS there only exist finitely many cycles $\mathcal{P}$. The automation of the technique is based on the generation of inequalities. For every cycle $\mathcal{P}$ we search for a well-founded quasi-ordering $\geq_{\mathcal{P}}$ satisfying $s \geq_{\mathcal{P}} t$ for all dependency pairs $\langle s, t \rangle$ in $\mathcal{P}$. Moreover, for at least one $\langle s, t \rangle$ in $\mathcal{P}$ we demand $s >_{\mathcal{P}} t$. In addition, to ensure $t\sigma \geq_{\mathcal{P}} v\sigma$ whenever $t\sigma$ reduces to $v\sigma$ (for consecutive pairs $\langle s, t \rangle$ and $\langle v, w \rangle$), we have to demand $l \geq_{\mathcal{P}} r$ for all those rules $l \to r$ of the TRS that may be used in this reduction. As we restrict ourselves to *normal* substitutions σ, not all rules are usable in a reduction of $t\sigma$. In general, if t contains a defined symbol f, then all f-rules are *usable* and moreover, all rules that are *usable* for right-hand sides of f-rules are also *usable* for t. Now we obtain the following theorem for automated[3] innermost termination proofs.

Theorem 2 (Innermost Termination Proofs). *A finite TRS is innermost terminating if for each cycle $\mathcal{P}$ there is a well-founded weakly monotonic quasi-ordering $\geq_{\mathcal{P}}$ where both $\geq_{\mathcal{P}}$ and $>_{\mathcal{P}}$ are closed under substitution, such that*

---

[3] Additional refinements for the automation can be found in [AG97b,AG99].

- $l \geq_{\mathcal{P}} r$ *for all rules $l \to r$ that are usable for some t with $\langle s, t \rangle \in \mathcal{P}$,*
- $s \geq_{\mathcal{P}} t$ *for all dependency pairs $\langle s, t \rangle$ from $\mathcal{P}$, and*
- $s >_{\mathcal{P}} t$ *for at least one dependency pair $\langle s, t \rangle$ from $\mathcal{P}$.*

Note that for Thm. 1 and 2 it is crucial to consider *all* cycles $\mathcal{P}$, not just the minimal ones (which contain no other cycles as proper subsets).

In Sect. 2 we presented the rules for the auxiliary functions in our example. Proving absence of infinite innermost chains for the cycles of their dependency pairs is very straightforward using Thm. 2. (So all library functions of our TRS are innermost terminating.) Moreover, as we assumed f to be a terminating function, its cycles do not lead to infinite innermost chains either.

Recall that (9)-(13) are not on cycles together with the remaining dependency pairs. Thus, what is left for verifying the desired property is proving absence of infinite innermost chains for the cycles $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$, where all rules of the whole TRS are possible candidates for being usable rules (also the rules for the unspecified function f).

Thm. 2 demands $s \geq_{\mathcal{P}} t$ resp. $s >_{\mathcal{P}} t$ for dependency pairs $\langle s, t \rangle$ on cycles. However for (9)-(13), these inequalities are not satisfied by any quasi-simplification ordering.[4] Thus, the automated proof fails here. Moreover, it is unclear which inequalities we have to add for the usable rules, since the rules for f are not given. Therefore, we have to extend the dependency pair technique.

5 Narrowing Dependency Pairs

To prove the absence of infinite innermost chains, for a dependency pair $\langle v, w \rangle$ it would be sufficient to demand $v\sigma \geq_{\mathcal{P}} w\sigma$ resp. $v\sigma >_{\mathcal{P}} w\sigma$ just for those instantiations σ where an instantiated right component $t\sigma$ of a previous dependency pair $\langle s, t \rangle$ reduces to $v\sigma$. For example, (11) only has to be regarded for instantiations σ where the instantiated right component $\mathsf{IF}_2(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, store)))\sigma$ of (10) reduces to the instantiated left component $\mathsf{IF}_2(store, m, \mathsf{false})\sigma$ of (11). In fact, this can only happen if *store* is not empty, i.e., if *store* reduces to the form $\mathsf{cons}(h, t)$. However, this observation has not been used in the inequalities of Thm. 2 and hence, we could not find an ordering for them. Thus, the idea is to perform the computation of empty on the level of the dependency pair. For that purpose the well-known concept of *narrowing* is extended to pairs of terms.

Definition 5 (Narrowing Pairs). *If a term t narrows to a term t' via the substitution μ, then the pair of terms $\langle s, t \rangle$ narrows to the pair $\langle s\mu, t' \rangle$.*

For example, the narrowings of the dependency pair (10) are

$$\langle \mathsf{IF}_1(x, 0, \mathsf{true}), \mathsf{IF}_2(x, 0, \mathsf{empty}(\mathsf{nil})) \rangle \tag{10a}$$
$$\langle \mathsf{IF}_1(\mathsf{nil}, \mathsf{s}(n), \mathsf{true}), \mathsf{IF}_2(\mathsf{nil}, \mathsf{s}(n), \mathsf{empty}(\mathsf{nil})) \rangle \tag{10b}$$
$$\langle \mathsf{IF}_1(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{true}), \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{empty}(\mathsf{cons}(h, \mathsf{fstsplit}(n, t)))) \rangle. \tag{10c}$$

---

[4] Essentially, the reason is that the left-hand side of dependency pair (9) is embedded in the right-hand sides of the pairs (11) and (13).

Thus, if a dependency pair $\langle s, t \rangle$ is followed by some dependency pairs $\langle v, w \rangle$ in an innermost chain and if t is not already unifiable with v (i.e., at least one rule is needed to reduce $t\sigma$ to $v\sigma$), then in order to 'approximate' the possible reductions of $t\sigma$ we may replace $\langle s, t \rangle$ by *all* its narrowings. Hence, we can replace the dependency pair (10) by the new pairs (10a)-(10c).

This enables us to extract necessary information from the last arguments of if's, i.e., from the former conditions of the CTRS. Thus, the narrowing refinement is the main reason why the transformation of CTRSs into TRSs is useful when analyzing the termination behaviour with dependency pairs. The number of narrowings for a pair is finite (up to variable renaming) and it can easily be computed automatically. The soundness of this technique is proved in [AG99].

Theorem 3 (Narrowing Refinement). *Let $\mathcal{P}$ be a set of pairs of terms and let $\langle s, t \rangle \in \mathcal{P}$ such that $Var(t) \subseteq Var(s)$ and such that for all (renamings of) $\langle v, w \rangle \in \mathcal{P}$, the terms t and v are not unifiable. Let $\mathcal{P}'$ result from $\mathcal{P}$ by replacing $\langle s, t \rangle$ by all its narrowings. If there exists no infinite innermost chain of pairs from $\mathcal{P}'$, then there exists no infinite innermost chain of pairs from $\mathcal{P}$ either.*

So we may always replace a dependency pair by all its narrowings. However, while this refinement is sound, in general it destroys the necessity of our innermost termination criterion in Thm. 1. For example, the TRS with the rules $\mathsf{f}(\mathsf{s}(x)) \to \mathsf{f}(\mathsf{g}(\mathsf{h}(x)))$, $\mathsf{g}(\mathsf{h}(x)) \to \mathsf{g}(x)$, $\mathsf{g}(0) \to \mathsf{s}(0)$, $\mathsf{h}(0) \to 1$ is innermost terminating. But if the dependency pair $\langle \mathsf{F}(\mathsf{s}(x)), \mathsf{F}(\mathsf{g}(\mathsf{h}(x))) \rangle$ is replaced by its narrowings $\langle \mathsf{F}(\mathsf{s}(0)), \mathsf{F}(\mathsf{g}(1)) \rangle$ and $\langle \mathsf{F}(\mathsf{s}(x)), \mathsf{F}(\mathsf{g}(x)) \rangle$, then $\langle \mathsf{F}(\mathsf{s}(x)), \mathsf{F}(\mathsf{g}(x)) \rangle$ forms an infinite innermost chain (using the instantiation $\{x/0\}$).

Nevertheless, in the application domain of process verification, we can restrict ourselves to *non-overlapping* TRSs. The following theorem shows that for these TRSs, narrowing dependency pairs indeed is a completeness preserving technique. More precisely, whenever innermost termination can be proved with the pairs $\mathcal{P}$, then it can also be proved with the pairs $\mathcal{P}'$.

Theorem 4 (Narrowing Dependency Pairs Preserves Completeness). *Let $\mathcal{R}$ be an innermost terminating non-overlapping TRS and let $\mathcal{P}$, $\mathcal{P}'$ be as in Thm. 3. If there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}$, then there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}'$ either.*

Proof. We show that every innermost $\mathcal{R}$-chain $\ldots \langle v_1, w_1 \rangle \langle s', t' \rangle \langle v_2, w_2 \rangle \ldots$ from $\mathcal{P}'$ can be transformed into an innermost chain from $\mathcal{P}$ of same length. There must be a substitution σ such that for all pairs the instantiated left-hand side is a normal form and the instantiated right-hand side reduces to the instantiated left-hand side of the next pair in the innermost chain. So in particular we have

$$w_1\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s'\sigma \ \text{ and } \ t'\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} v_2\sigma.$$

We know that $\langle s, t \rangle$ narrows to $\langle s', t' \rangle$ via a substitution μ. As the variables in $\langle s, t \rangle$ are disjoint from all other variables, we may extend σ to 'behave' like $\mu\sigma$ on the variables of s and t. Then we have $s\sigma = s\mu\sigma = s'\sigma$ and hence, $w_1\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s\sigma$.

Moreover, by the definition of narrowing, $t\mu \to_\mathcal{R} t'$. This implies $t\mu\sigma \to_\mathcal{R} t'\sigma$ and as $t\sigma = t\mu\sigma$, we have $t\sigma \to_\mathcal{R} t'\sigma \overset{i}{\to_\mathcal{R}^*} v_2\sigma$ where $v_2\sigma$ is a normal form. As $\mathcal{R}$ is innermost terminating and non-overlapping, it is convergent. Thus, every term has a unique normal form and hence, repeated application of *innermost* reduction steps to $t\sigma$ also yields the normal form $v_2\sigma$, i.e., $t\sigma \overset{i}{\to_\mathcal{R}^*} v_2\sigma$.
Thus, ... $\langle v_1, w_1 \rangle \langle s, t \rangle \langle v_2, w_2 \rangle$ ... is also an innermost $\mathcal{R}$-chain. $\qquad\square$

Hence, *independent* of the technique used to check the absence of infinite innermost chains, narrowing dependency pairs can never destroy the success of the innermost termination proof. Moreover, narrowing can of course be repeated an *arbitrary* number of times. Thus, after replacing (10) by (10a)-(10c), we may subsequently replace (10a) and (10b) by their respective narrowings.

$$\langle \mathsf{IF}_1(x, 0, \mathsf{true}), \mathsf{IF}_2(x, 0, \mathsf{true}) \rangle \tag{10aa}$$
$$\langle \mathsf{IF}_1(\mathsf{nil}, \mathsf{s}(n), \mathsf{true}), \mathsf{IF}_2(\mathsf{nil}, \mathsf{s}(n), \mathsf{true}) \rangle \tag{10ba}$$

This excludes them from being on a cycle in the estimated innermost dependency graph. Thus, now instead of the dependency pairs (9)-(13) we consider (9), (10c), (11), (12), and (13). A further narrowing of (10c) is not necessary for our purposes (but according to Thm. 4 it would not harm either). The right component of the dependency pair (11) unifies with the left component of (9) and therefore, (11) must not be narrowed. Instead we narrow (9).

$$\langle \mathsf{PROCESS}(\mathsf{nil}, m), \mathsf{IF}_1(\mathsf{nil}, m, \mathsf{leq}(m, 0)) \rangle \tag{9a}$$
$$\langle \mathsf{PROCESS}(\mathsf{cons}(h, t), m), \mathsf{IF}_1(\mathsf{cons}(h, t), m, \mathsf{leq}(m, \mathsf{s}(\mathsf{length}(t)))) \rangle \tag{9b}$$
$$\langle \mathsf{PROCESS}(store, 0), \mathsf{IF}_1(store, 0, \mathsf{true}) \rangle \tag{9c}$$

By narrowing (10) to (10c), we determined that we only have to regard instantiations where *store* has the form $\mathsf{cons}(h, t)$ and m has the form $\mathsf{s}(n)$. Thus, (9a) and (9c) do not occur on a cycle and therefore, (9) can be replaced by (9b) only.

As (11)'s right component does not unify with left components any longer, we may now narrow (11) as well. By repeated narrowing steps and by dropping those pairs which do not occur on cycles, (11) can be replaced by

$$\langle \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(n, t), \mathsf{s}(n)) \rangle \tag{11aac}$$
$$\langle \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{app}(\mathsf{nil}, \mathsf{sndsplit}(n, t)), \mathsf{s}(n)) \rangle \tag{11ad}$$
$$\langle \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}(n, t)), \mathsf{s}(n)) \rangle \tag{11d}$$

Now for the cycle $\mathcal{P}_1$, it is (for example) sufficient to demand that (11aac), (11ad), and (11d) are strictly decreasing and that (9b), (10c), and all usable rules are weakly decreasing. Similar narrowings can also be applied for the pairs (12) and (13) which results in analogous inequalities for the cycles $\mathcal{P}_2$ and $\mathcal{P}_3$.

Most standard orderings amenable to automation are *strongly* monotonic path orderings (cf. e.g. [Der87,Ste95]), whereas here we only need *weak* monotonicity. Hence, before synthesizing a suitable ordering, some of the arguments of function symbols may be eliminated, cf. [AG99]. For example, in our inequalities one may eliminate the third argument of IF_2. Then every term $\mathsf{IF}_2(t_1, t_2, t_3)$ in the inequalities is replaced by $\mathsf{IF}_2'(t_1, t_2)$ (where IF_2' is a new binary function

symbol). By comparing the terms resulting from this replacement instead of the original terms, we can take advantage of the fact that IF_2 does not have to be strongly monotonic in its third argument. Similarly, in our example we will also eliminate the third arguments of IF_1 and IF_3 and the first argument of $\mathsf{sndsplit}$. Note that there are only finitely many (and only few) possibilities to eliminate arguments of function symbols. Therefore all these possibilities can be checked automatically. In this way, the recursive path ordering (rpo) satisfies the inequalities for (11aac), (9b), (10c), for the dependency pairs resulting from (12) and (13), and for all (known) usable rules. However, the inequalities resulting from (11ad) and (11d)

$$\mathsf{IF}_2'(\mathsf{cons}(h, t), \mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{app}(\mathsf{nil}, \mathsf{sndsplit}'(t)), \mathsf{s}(n))$$
$$\mathsf{IF}_2'(\mathsf{cons}(h, t), \mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{app}(\mathsf{map\_f}(\mathsf{self}, \mathsf{nil}), \mathsf{sndsplit}'(t)), \mathsf{s}(n))$$

are not satisfied because of the app-terms on the right-hand sides (as the app-rule forces app to be greater than cons in the precedence of the rpo). Moreover, the map_f-term in the inequalities requires us to consider the usable rules corresponding to the (unspecified) Erlang function f as well.

To get rid of these terms, one would like to perform narrowing on map_f and app. However, in general narrowing only *some* subterms of right components is unsound.[5] Instead, we always have to replace a pair by *all* its narrowings. But then narrowing (11ad) and (11d) provides no solution here, since narrowing the sndsplit-subterm results in pairs containing problematic app- and map_f-terms again. In the next section we describe a technique which solves the above problem.

6   Rewriting Dependency Pairs

While performing only some *narrowing* steps is unsound, for non-overlapping TRSs it is at least sound to perform only one of the possible *rewrite* steps.[6] So if $t \to r$, then we may replace a dependency pair $\langle s, t \rangle$ by $\langle s, r \rangle$. Note that this technique is only applicable to *dependency pairs*, but not to *rules* of the TRS. Indeed, by reducing the right-hand side of a rule, a non (innermost) terminating TRS can be transformed into a terminating one, even if the TRS is non-overlapping. As an example regard the TRS with the rules $0 \to \mathsf{f}(0)$, $\mathsf{f}(x) \to 1$ which is clearly not innermost terminating. However, if the right-hand side of the first rule is rewritten to 1, then the resulting TRS is terminating. The following theorem proves that our refinement of the dependency pair approach is sound.

---

[5] As an example regard the TRS $\mathsf{f}(0, 1) \to \mathsf{s}(1)$, $\mathsf{f}(x, 0) \to 1$, $\mathsf{a} \to 0$, and $\mathsf{g}(\mathsf{s}(y)) \to \mathsf{g}(\mathsf{f}(\mathsf{a}, y))$. If we would replace the dependency pair $\langle \mathsf{G}(\mathsf{s}(y)), \mathsf{G}(\mathsf{f}(\mathsf{a}, y)) \rangle$ by only one of its narrowings, viz. $\langle \mathsf{G}(\mathsf{s}(0)), \mathsf{G}(1) \rangle$, then one could falsely prove innermost termination, although the term $\mathsf{g}(\mathsf{s}(1))$ starts an infinite innermost reduction.

[6] Combining narrowing and rewriting is common in *normal narrowing* strategies to solve E-unification problems [Fay79,Han94]. However, in contrast to our approach, *normal narrowing* is only used for convergent TRSs and instead of performing one (or arbitrary) many rewrite steps, there one rewrites terms to normal forms.

Theorem 5 (Rewriting Dependency Pairs). *Let $\mathcal{R}$ be non-overlapping and let $\mathcal{P}$ be a set of pairs of terms. Let $\langle s, t \rangle \in \mathcal{P}$, let $t \to_{\mathcal{R}} r$ and let $\mathcal{P}'$ result from $\mathcal{P}$ by replacing $\langle s, t \rangle$ by $\langle s, r \rangle$. If there exists no infinite innermost chain of pairs from $\mathcal{P}'$, then there exists no infinite innermost chain from $\mathcal{P}$ either.*

Proof. By replacing all (renamed) occurrences of $\langle s, t \rangle$ with the corresponding renamed occurrences of $\langle s, r \rangle$, every innermost chain $\ldots \langle s, t \rangle \langle v, w \rangle \ldots$ from $\mathcal{P}$ can be translated into an innermost chain from $\mathcal{P}'$ of same length. The reason is that there must be a substitution σ with $t\sigma \stackrel{\mathsf{i}}{\to}_{\mathcal{R}}^* v\sigma$ where $v\sigma$ is a normal form. So $t\sigma$ is weakly innermost normalizing and thus, by [Gra96a, Thm. 3.2.11 (1a) and (4a)], $t\sigma$ is confluent and strongly normalizing. With $t \to_{\mathcal{R}} r$, we obtain $t\sigma \to_{\mathcal{R}} r\sigma$. Hence, $r\sigma$ is strongly normalizing as well and thus, it also reduces innermost to some normal form q. Now confluence of $t\sigma$ implies $q = v\sigma$. Therefore, $\ldots \langle s, r \rangle \langle v, w \rangle \ldots$ is an innermost chain, too. $\qquad\square$

The converse of Thm. 5 holds as well if $\mathcal{P}$ is obtained from the dependency pairs by repeated narrowing and rewriting steps. So similar to *narrowing*, *rewriting* dependency pairs does not destroy the necessity of our criterion either.

Theorem 6 (Rewriting Dependency Pairs Preserves Completeness). *Let $\mathcal{R}$ be an innermost terminating non-overlapping TRS and let $\mathcal{P}$, $\mathcal{P}'$ be as in Thm. 5. If there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}$, then there exists no infinite innermost $\mathcal{R}$-chain of pairs from $\mathcal{P}'$ either.*

Proof. In an innermost chain $\ldots \langle s, r \rangle \langle v, w \rangle \ldots$ from $\mathcal{P}'$, replacing all (renamed) occurrences of $\langle s, r \rangle$ by corresponding renamings of $\langle s, t \rangle$ yields an innermost chain from $\mathcal{P}$ of same length. The reason is that there must be a σ with $r\sigma \stackrel{\mathsf{i}}{\to}_{\mathcal{R}}^* v\sigma$. Thus, $t\sigma \to_{\mathcal{R}} r\sigma \stackrel{\mathsf{i}}{\to}_{\mathcal{R}}^* v\sigma$ implies $t\sigma \stackrel{\mathsf{i}}{\to}_{\mathcal{R}}^* v\sigma$ by the convergence of $\mathcal{R}$. $\qquad\square$

In our example we may now eliminate app and map_f by rewriting the pairs (11ad) and (11d). Even better, before narrowing, we could first rewrite (11), (12), and (13). Moreover, we could simplify (10c) by rewriting it as well. Thus, the resulting pairs on the cycles we are interested in are:

$$\langle \mathsf{PROCESS}(\mathsf{cons}(h, t), m), \mathsf{IF}_1(\mathsf{cons}(h, t), m, \mathsf{leq}(m, \mathsf{s}(\mathsf{length}(t)))) \rangle \quad (9\mathrm{b})$$
$$\langle \mathsf{IF}_1(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{true}), \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}) \rangle \quad (10\mathrm{c}')$$
$$\langle \mathsf{IF}_2(store, m, \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(m, store), m) \rangle \quad (11')$$
$$\langle \mathsf{IF}_1(store, m, \mathsf{false}), \mathsf{IF}_3(store, m, \mathsf{empty}(\mathsf{fstsplit}(m, store))) \rangle \quad (12')$$
$$\langle \mathsf{IF}_3(store, m, \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(m, store), m) \rangle \quad (13')$$

Analogous to Sect. 5, now we narrow $(11')$, $(12')$, $(13')$, perform a rewrite step for one of $(12')$'s narrowings, and delete those resulting pairs which are not on any cycle. In this way, $(11')$, $(12')$, $(13')$ are replaced by

$$\langle \mathsf{IF}_2(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(n, t), \mathsf{s}(n)) \rangle \quad (11'')$$
$$\langle \mathsf{IF}_1(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{IF}_3(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}) \rangle \quad (12'')$$
$$\langle \mathsf{IF}_3(\mathsf{cons}(h, t), \mathsf{s}(n), \mathsf{false}), \mathsf{PROCESS}(\mathsf{sndsplit}(n, t), \mathsf{s}(n)) \rangle \quad (13'')$$

13

By eliminating the first argument of sndsplit and the third arguments of IF_1, IF_2, and IF_3 (cf. Sect. 5), we obtain the following inequalities. Note that according to Thm. 2, these inequalities prove the absence of infinite innermost chains for all three cycles built from (9b), (10c$'$), and (11$''$)-(13$''$), since for each of these cycles (at least) one of its dependency pairs is strictly decreasing.

$$\mathsf{PROCESS}(\mathsf{cons}(h,t),m) \geq \mathsf{IF}'_1(\mathsf{cons}(h,t),m) \qquad\qquad \mathsf{sndsplit}'(x) \geq x$$
$$\mathsf{IF}'_1(\mathsf{cons}(h,t),\mathsf{s}(n)) \geq \mathsf{IF}'_2(\mathsf{cons}(h,t),\mathsf{s}(n)) \qquad\qquad \mathsf{sndsplit}'(\mathsf{nil}) \geq \mathsf{nil}$$
$$\mathsf{IF}'_1(\mathsf{cons}(h,t),\mathsf{s}(n)) \geq \mathsf{IF}'_3(\mathsf{cons}(h,t),\mathsf{s}(n)) \qquad \mathsf{sndsplit}'(\mathsf{cons}(h,t)) \geq \mathsf{sndsplit}'(t)$$
$$\mathsf{IF}'_2(\mathsf{cons}(h,t),\mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{sndsplit}'(t),\mathsf{s}(n)) \qquad l \geq r \text{ for all rules } l \to r$$
$$\mathsf{IF}'_3(\mathsf{cons}(h,t),\mathsf{s}(n)) > \mathsf{PROCESS}(\mathsf{sndsplit}'(t),\mathsf{s}(n)) \qquad \text{with } root(l) \in \{\mathsf{leq}, \mathsf{length}\}$$

Now these inequalities are satisfied by the rpo. The right column contains all inequalities corresponding to the usable rules, since the rules for map_f and f are no longer usable. Hence, the TRS of Sect. 3 is innermost terminating. In this way, left-right decreasingness of the CTRS from Sect. 2 could be proved automatically. Therefore, the desired property holds for the original Erlang process.

7 Conclusion

We have shown that rewriting techniques (and in particular, the dependency pair approach) can be successfully applied for process verification tasks in industry. While our work was motivated by a specific process verification problem, in this paper we developed several new techniques which are of general use in term rewriting. First of all, we showed how dependency pairs can be utilized to prove that *conditional* term rewriting systems are decreasing and terminating. Moreover, we presented two refinements which considerably increase the class of systems where dependency pairs are successful. The first refinement of *narrowing* dependency pairs was already introduced in [AG99], but completeness of the technique for non-overlapping TRSs is a new result. It ensures that application of the narrowing technique can never destroy the success of such an innermost termination proof. In fact, our narrowing refinement is the main reason why the approach of handling CTRSs by transforming them into TRSs is successful in combination with the dependency pair approach (whereas this transformation is usually not of much use for the standard termination proving techniques). Finally, to strengthen the power of dependency pairs we introduced the novel technique of *rewriting* dependency pairs and proved its soundness and completeness for innermost termination of non-overlapping TRSs.

References

[AD99]   T. Arts & M. Dam, Verifying a distributed database lookup manager written in Erlang. In *Proc. FM '99*, Toulouse, France, 1999.

[AG97a] T. Arts & J. Giesl, Automatically proving termination where simplification orderings fail. *TAPSOFT '97*, LNCS 1214, pp. 261–273, Lille, France, 1997.

[AG97b] T. Arts & J. Giesl, Proving innermost normalisation automatically. In *Proc. RTA-97*, LNCS 1232, pp. 157–172, Sitges, Spain, 1997.

[AG98] T. Arts & J. Giesl, Modularity of termination using dependency pairs. In *Proc. RTA-98*, LNCS 1232, pp. 226–240, Tsukuba, Japan, 1998.

[AG99] T. Arts & J. Giesl, Termination of term rewriting using dependency pairs. *TCS*. To appear. Preliminary version under http://www.inferenzsysteme. informatik.tu-darmstadt.de/~reports/notes/ibn-97-46.ps

[BN98] F. Baader & T. Nipkow, *Term Rewriting and All That.* Cambridge University Press, 1998.

[BK86] J. A. Bergstra & J. W. Klop, Conditional rewrite rules: confluence and termination. *JCSS*, 32:323–362, 1986.

[BG89] H. Bertling & H. Ganzinger, Completion-time optimization of rewrite-time goal solving. *Proc. RTA-89*, LNCS 355, pp. 45–58, Chapel Hill, USA, 1989.

[DP87] N. Dershowitz & D. A. Plaisted, Equational programming. *Machine Intelligence*, 11:21–56, Oxford University Press, 1987.

[Der87] N. Dershowitz, Termination of rewriting. *JSC*, 3:69–116, 1987.

[DOS88] N. Dershowitz, M. Okada, & G. Sivakumar, Canonical conditional rewrite systems. In *Proc. CADE-9*, LNCS 310, pp. 538–549, Argonne, USA, 1988.

[DO90] N. Dershowitz & M. Okada, A rationale for conditional equational programming. *TCS*, 75:111–138, 1990.

[DJ90] N. Dershowitz & J.-P. Jouannaud, Rewrite Systems. In *Handbook of Theoretical Computer Science, Vol. B*, pp. 243-320, Elsevier, 1990.

[DH95] N. Dershowitz & C. Hoot, Natural termination. *TCS*, 142(2):179–207, 1995.

[Fay79] M. J. Fay, First-order unification in an equational theory. *Proc. 4th Workshop on Automated Deduction*, pp. 161-167, Austin, TX, Academic Press, 1979.

[GM87] E. Giovanetti & C. Moiso, Notes on the eliminations of conditions. In *Proc. CTRS '87*, LNCS 308, pp. 91–97, Orsay, France, 1987.

[Gra94] B. Gramlich, On termination and confluence of conditional rewrite systems. In *Proc. CTRS '94*, LNCS 968, pp. 166–185, Jerusalem, Israel, 1994.

[Gra95] B. Gramlich, Abstract relations between restricted termination and confluence properties of rewrite systems. *Fundamenta Informaticae*, 24:3–23, 1995.

[Gra96a] B. Gramlich, *Termination and confluence properties of structured rewrite systems*. PhD Thesis, Universität Kaiserslautern, Germany, 1996.

[Gra96b] B. Gramlich, On termination and confluence properties of disjoint and constructor-sharing conditional rewrite systems. *TCS*, 165:97–131, 1996.

[Han94] M. Hanus, The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19,20:583-628, 1994.

[HN99] Patent pending, Ericsson Telecom AB, 1999.

[Kap84] S. Kaplan, Conditional rewrite rules. *TCS*, 33:175–193, 1984.

[Mar96] M. Marchiori, Unravelings and Ultra-properties, *Proc. ALP '96*, LNCS 1139, pp. 107-121, Aachen, Germany, 1996.

[Mid93] A. Middeldorp, Modular properties of conditional term rewriting systems. *Information and Computation*, 104:110–158, 1993.

[Ste95] J. Steinbach, Simplification orderings: history of results. *Fundamenta Informaticae*, 24:47–87, 1995.

[SMI95] T. Suzuki, A. Middeldorp, & T. Ida, Level-confluence of conditional rewrite systems with extra variables in right-hand sides. *Proc. RTA-95*, LNCS 914, pp. 179–193, Kaiserslautern, Germany, 1995.

[WG94] C.-P. Wirth & B. Gramlich, A constructor-based approach for positive/negative conditional equational specifications. *JSC*, 17:51–90, 1994.