Shallow Dependency Pairs

Alexander Krauss

Technische Universität München, Institut für Informatik
http://www.in.tum.de/~krauss

Abstract. We show how the dependency pair approach, commonly used to modularize termination proofs of rewrite systems, can be adapted to establish termination of recursive functions in a system like Isabelle/HOL or Coq. It turns out that all that is required are two simple lemmas about wellfoundedness.

1 Introduction

Termination proofs are essential in theorem proving, as they are required to justify the definition of recursive functions.

Dependency pairs [1,8] are currently one of the most successful approaches to prove termination of term rewrite systems (TRSs). They have been successfully adapted to other situations like functional programs [7]. So it seems natural to try to use such techniques in a theorem proving context.

However, there are two main obstacles that make an adoption difficult: First, formalizing the underlying theory is a major technical effort, since the proofs are often quite technical and non-trivial, and involve a large variety of specialized concepts.

Second, there is a significant gap between the very syntax-centric view of term rewriting ("prove that a given list of rewrite rules over a certain signature allows no infinite rewrite sequences") and the more semantic view that we need here ("prove that a given relation over, say, natural numbers is wellfounded"), where terms, signatures and reductions never occur on the object level.

The first obstacle has been attacked by two recent formalization efforts [2,5], where large parts of the underlying metatheory were formally verified. This approach allows the certification of TRS proofs in an interactive proof assistant, but it cannot be used to justify the definition of recursive functions yet.

In this paper we are concerned with the second obstacle. We take the slightly more abstract relational view that is appropriate to deal with the termination proof obligations arising from function definitions. It turns out that some technical issues disappear in this setting, and that dependency pair proofs are nothing more than a clever application of the following two lemmas:

Lemma 1. $\mathit{wf}\ R \Longrightarrow \mathit{wf}\ S \Longrightarrow R \circ S \subseteq R \Longrightarrow \mathit{wf}\ (R \cup S)$

Lemma 2. $\mathit{wf}\ (R \cup S) = \mathit{wf}\ ((R \cup S) \circ R \cup S)$

This paper is structured as follows: After introducing some basic notions and notation in §2, we revisit briefly the traditional dependency pair approach in §3. In §4, we explain dependency pair proofs in the setting of Isabelle/HOL, and give some simple examples in §5 and §6.

2 Preliminaries

We work in Isabelle/HOL [12], but as we do not rely on any special features, the same ideas also apply in a system with different foundations, such as Coq. All theorems and proofs presented in this paper were mechanically checked by Isabelle.

2.1 Notation

In our notation, relations are represented as sets of pairs, and composition is defined as

$$R \circ S = \{(x,\, z) \mid \exists\, y.\ (x,\, y) \in R \wedge (y,\, z) \in S\}\ .$$

We will frequently write comprehensions of the form $\{f\ x\ y\ |x\ y.\ P\ x\ y\}$, which abbreviates $\{\ u.\ \exists\, x\ y.\ u = f\ x\ y \wedge P\ x\ y\}$.

2.2 Termination Proof Obligations

Consider the following example function:

$foo\ n\ 0 = n$
$foo\ 0\ (Suc\ m) = foo\ (Suc\ m)\ (Suc\ m)$
$foo\ (Suc\ n)\ (Suc\ m) = foo\ n\ m$

When a recursive function is defined in Isabelle, the system automatically produces a proof obligation that corresponds to the termination of the function (cf. [10,13]). This proof obligation states the wellfoundedness of the function's call relation (which relates each possible argument of the function with the resulting recursive calls). For the above definition, we obtain the goal

$$\begin{aligned}&wf\ (\{((Suc\ m,\ Suc\ m),\ (0,\ Suc\ m))\ |m.\ True\} \cup\\ &\quad \{((n,\ m),\ (Suc\ n,\ Suc\ m))\ |m\ n.\ True\})\end{aligned}$$

Note how each recursive call is written as a relation comprehension. Isabelle's standard definition of wellfoundedness follows the convention that the "smaller" element in a relation appears on the left. Consequently, the arguments from the left hand sides of the equations above must go to the right and vice-versa.

In previous work (e.g. Bulwahn et al. [3]), this proof obligation was sometimes presented differently, asking for an embedding into another wellfounded relation:

1. wf ?R
2. $\bigwedge m.\ ((Suc\ m,\ Suc\ m),\ (0,\ Suc\ m)) \in\ ?R$
3. $\bigwedge n\ m.\ ((n,\ m),\ (Suc\ n,\ Suc\ m)) \in\ ?R$

In this paper, we prefer the first version, which is equivalent, but gives us more flexibility to modify the problem. It is easy to convert between one form of the goal to the other.

The general form of the proof obligation is the following:

$$\begin{aligned}
&wf\ (\{\ (r_1,\ l_1)\ \mid\ v_1\ldots v_{m_1}.\ \Gamma_1\ \}\\
&\quad\cup \ldots\\
&\quad\cup \ldots\\
&\quad\cup \{\ (r_n,\ l_n)\ \mid\ v_1\ldots v_{m_n}.\ \Gamma_n\ \})
\end{aligned}$$

Thus we must prove wellfoundedness of a relation that is given as a union of relation comprehensions, each reflecting a recursive call from l_k to r_k under the conditions Γ_k. Here, the bound variables $v_1,\ \ldots,\ v_m$ can occur in r, l and Γ.

If we take the relations C and D corresponding to two recursive calls, we can form their composition $D \circ C$, which expresses the call C which is immediately followed by D[1]. Such a composition can again be written as a comprehension:

$$\begin{aligned}
&\{(a\ x,\ b\ x)\ |x.\ P\ x\} \circ \{(c\ y,\ d\ y)\ |y.\ Q\ y\} =\\
&\{(a\ x,\ d\ y)\ |x\ y.\ b\ x = c\ y \wedge P\ x \wedge Q\ y\}
\end{aligned}$$

3 Dependency Pairs in Rewriting

We give a very brief (and slightly oversimplified) introduction to dependency pairs as they are used in termination proofs for term rewriting. The readers who are interested in more details should consult the original literature [1,8].

We consider first order terms over a fixed signature. Given a term rewrite system $\mathcal{R}$, i.e. a set of rules of the form $l \to r$, we say that a function symbol is *defined*, if it occurs in root position on the left hand side of some rule in $\mathcal{R}$. The set of *dependency pairs* (DPs) of $\mathcal{R}$ is defined as

$$\begin{aligned}
DP(\mathcal{R}) = \{l^\sharp \to t^\sharp\ \mid\ & l \to r \in \mathcal{R},\\
& t \text{ is a subterm of } r \text{ with a defined root symbol}
\end{aligned}$$

Here, $t^\sharp$ is just the term t whose root symbol is marked with a $^\sharp$, to distinguish it from a normal function symbol. Thus, dependency pairs just capture the usual notion of a recursive call.

A (finite or infinite) sequence of dependency pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ is called a *chain*, if there is a substitution σ, such that $\sigma(t_i) \to^*_{\mathcal{R}} \sigma(s_{i+1})$ for all

[1] It is a bit counterintuitive that the composition is written in the “wrong” order. Again, this comes from the fact that the smaller element is written left.

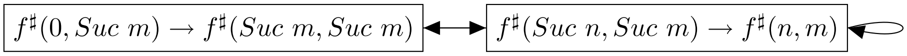
i. Such chains model sequences of recursive calls as they can occur during a reduction. A TRS terminates if and only if there are no infinite chains.

The *dependency graph* of a TRS $\mathcal{R}$ is the graph with the nodes $DP(\mathcal{R})$ and an edge between two DPs, if they form a chain (of length two). The dependency graph serves as a simple model of control flow and allows the decomposition of a termination problem into smaller parts: if it has more than one strongly connected component (SCC), these can be split apart and dealt with separately.

The resulting smaller problems can be tackled using orderings: for a pair of relations $(\prec, \preceq)$ with certain properties[2] (called a *reduction pair*), we can remove all DPs that respect $\succ$, if the remaining DPs respect $\succeq$. Note that after removing DPs like this, the dependency graph may again fall apart into different SCCs. We continue with the repeated decomposition and application of reduction pairs until we reach the empty set of DPs.

It is undecidable in general whether two DPs form a chain. So the dependency graph is usually approximated using some safe heuristic [1]. The most commonly used heuristic simply checks if the right hand side of one DP and the left hand side of the other DP are unifiable after replacing all defined symbols by fresh variables and renaming the variables apart.

Example We now prove termination of *foo* using dependency pairs. The dependency pairs are the two recursive calls, and the dependency graph looks like this:


$f^\sharp(0, Suc\ m) \to f^\sharp(Suc\ m, Suc\ m)$
$f^\sharp(Suc\ n, Suc\ m) \to f^\sharp(n, m)$

There is only one SCC, so we cannot split the problem into pieces. Now, we note that the second argument is decreasing in the second call, and it stays the same in the first one[3]. This allows us to remove the second call, and just one dependency pair remains:

$$\boxed{f^\sharp(0, Suc\ m) \to f^\sharp(Suc\ m, Suc\ m)}$$

Since there is no edge here, there cannot possibly be a loop, and we are done. This step is remarkable, since we do not need to argue about decreasing measures and the like. It is enough to know that the call cannot happen twice in a row.

Dependency pair proofs do not yield a simple characterization of a wellfounded order compatible with all recursive calls. However, this is not required, as we will see in the next section.

[2] In addition to $\prec$ being wellfounded, they must be closed under substitutions, $\preceq$ must be closed under contexts and $\prec \circ \preceq \subseteq \prec$ and $\succeq \circ \succ \subseteq \succ$. Our adapted version in §4 will drop some of these requirements.

[3] In rewriting this is formalized with a subterm ordering and a so-called *argument filtering* that selects the second argument.

4 Shallow Dependency Pairs in HOL

We now see how we can "mimic" dependency pair proofs in a very shallow way, to solve proof obligations of the type given in §2. The relation comprehensions for the different recursive calls will take the place of the dependency pairs.

Before we start, note a trivial consequence of Lemma 2 (with $S = \emptyset$):

Corollary 1. $\mathit{wf}\, R = \mathit{wf}\ (R \circ R)$

4.1 Dependency Graph

Our dependency graph has an edge from call C to call D (each represented by a relation), iff C can be followed by D, i.e. iff

$D \circ C \neq \emptyset$

Like in §3, this property is undecidable, but we can safely approximate the dependency graph by drawing an edge whenever we are unable to find an proof of $C \circ D = \emptyset$ using some automated tactic we have at hand.

Whenever we have more than one strongly connected component, we can apply a divide-and-conquer strategy: Let R and S be sets of calls, such that no call from R can happen after some call from S. This means that $R \circ S = \emptyset \subseteq R$ holds, and we can apply Lemma 1 to split the two sets of calls into independent sub-problems. This step can be repeated until the graph is split into its SCCs.

4.2 Removing trivial SCCs

With the decomposition steps above, we might finally end up with the trivial SCC that just consists of a single call C that is not reachable from itself. This corresponds to a graph with one node and no edges.

Formally, since there is no edge, we have $C \circ C = \emptyset$. Using Corollary 1, we can rewrite the goal to $\mathit{wf}\ (C \circ C)$, which is trivial, since the empty set is always wellfounded.

4.3 Using reduction pairs

When problem cannot be simplified any more, some real progress needs to be made. If one of the calls strictly decreases with respect to a certain ordering, and all the other calls are weakly decreasing ("less or equal"), then that call can be removed.

The notion of strict and weak decrease wrt. arbitrary orderings is expressed by a reduction pair, that is a pair of relations $R_<$ and $R_\leq$ satisfying:

$\mathit{reduction\text{-}pair}\ (R_<,\, R_\leq) = (\mathit{wf}\, R_< \wedge R_< \circ R_\leq \subseteq R_<)$

The canonical example of a reduction pair is a quasi-order, whose strict part is wellfounded, but the above definition is more general, since transitivity is not required. We usually construct reduction pairs from measure functions into the natural numbers by taking the inverse images of $<$ and $\leq$.

The following is a direct consequence of Lemma 1 and the above definition:

Corollary 2. *reduction-pair* $(R_<, R_\leq) \Longrightarrow S \subseteq R_< \Longrightarrow S' \subseteq R_\leq \Longrightarrow$ *wf* S' $\Longrightarrow$ *wf* $(S \cup S')$

Operationally, we are allowed to remove some calls S from a proof obligation, if we can find a reduction pair $(R_<, R_\leq)$, such that S can be embedded into $R_<$ and S' can be embedded into $R_\leq$.

Of course, *finding* useful reduction pairs can be very difficult in general, but in practice many useful relations can be found by very simple heuristics. Basically, we can use the same heuristics as in [3].

Note that after removing a call, the dependency graph may again fall apart into different SCCs.

5 Examples

Given these tools, the termination proof for *foo* is easy. We start with the goal we already saw in §2:

$$\mathit{wf}\ (\{((\mathit{Suc}\ m,\ \mathit{Suc}\ m),\ (0,\ \mathit{Suc}\ m))\ |m.\ \mathit{True}\} \cup$$
$$\{((n,\ m),\ (\mathit{Suc}\ n,\ \mathit{Suc}\ m))\ |m\ n.\ \mathit{True}\})$$

First, we can attack the second call using the reduction pair given by the measure function *snd*. Proving strict and weak descent of the second and the first call, respectively, is simple. Hence, applying Corollary 2, we get rid of that call, and are left with just the goal

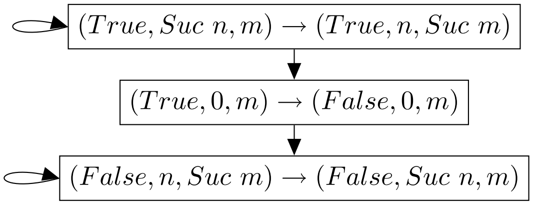
1. $\mathit{wf}\ \{((\mathit{Suc}\ m,\ \mathit{Suc}\ m),\ (0,\ \mathit{Suc}\ m))\ |m.\ \mathit{True}\}$

This SCC is trivial, since we can easily show that $C \circ C = \emptyset$ for this call, which concludes the proof.

Let us look at another (artificial) example. The function *bar* has two modes. Depending on a boolean flag, either the first or the second argument decreases:

bar True $(\mathit{Suc}\ n)\ m = \mathit{bar}\ \mathit{True}\ n\ (\mathit{Suc}\ m)$
bar True $0\ m = \mathit{bar}\ \mathit{False}\ 0\ m$
bar False $n\ (\mathit{Suc}\ m) = \mathit{bar}\ \mathit{False}\ (\mathit{Suc}\ n)\ m$
bar False $n\ 0 = n$

This specific control flow becomes clearly visible in the dependency graph:


$(True, Suc\ n, m) \to (True, n, Suc\ m)$
$(True, 0, m) \to (False, 0, m)$
$(False, n, Suc\ m) \to (False, Suc\ n, m)$

Now, we apply the above techniques to prove termination. As usual, we start with the goal:

$1.\ wf\ (\{((True,\ n,\ Suc\ m),\ (True,\ Suc\ n,\ m))\ |n\ m.\ True\} \cup$
$\quad\quad \{((False,\ 0,\ m),\ (True,\ 0,\ m))\ |m.\ True\} \cup$
$\quad\quad \{((False,\ Suc\ n,\ m),\ (False,\ n,\ Suc\ m))\ |n\ m.\ True\})$

Using Lemma 1, we can split off the first call, since it can never follow one of the other calls:

$1.\ wf\ \{((True,\ n,\ Suc\ m),\ (True,\ Suc\ n,\ m))\ |n\ m.\ True\}$
$2.\ wf\ (\{((False,\ 0,\ m),\ (True,\ 0,\ m))\ |m.\ True\} \cup$
$\quad\quad \{((False,\ Suc\ n,\ m),\ (False,\ n,\ Suc\ m))\ |n\ m.\ True\})$

The other two calls can be split in the same way, and we have three independent sub-problems:

$1.\ wf\ \{((True,\ n,\ Suc\ m),\ (True,\ Suc\ n,\ m))\ |n\ m.\ True\}$
$2.\ wf\ \{((False,\ 0,\ m),\ (True,\ 0,\ m))\ |m.\ True\}$
$3.\ wf\ \{((False,\ Suc\ n,\ m),\ (False,\ n,\ Suc\ m))\ |n\ m.\ True\}$

Now that we have isolated them, each call can be solved on its own, and each in a different way: The first call is decreasing in the second argument, the second one is trivial, by Corollary 1, and the third one again has a measure, this time the third argument.

6 Merging

Sometimes, recursive calls occur on arguments that are not decreasing with respect to any obvious relation. But the increase might only be temporary. These functions are usually difficult to handle.

The following example occurs in a reflected arithmetic decision procedure formalized by Chaieb [4]. It operates on a datatype representing numeric expressions:

datatype *num* = *C int* | *Bound nat* | *CN nat int num* | *Neg num* | *Add num num*
| *Sub num num* | *Mul int num* | *Floor num* | *CF int num num*

The function *simpnum* simplifies *num* expressions, using the helper functions *numneg*, *numadd*, whose definitions we omit. Here is the equations for *simpnum*:

$\mathit{simpnum}\ (C\ j) = C\ j$
$\mathit{simpnum}\ (\mathit{Bound}\ n) = \mathit{CN}\ n\ 1\ (C\ 0)$
$\mathit{simpnum}\ (\mathit{Neg}\ t) = \mathit{numneg}\ (\mathit{simpnum}\ t)$
$\mathit{simpnum}\ (\mathit{Add}\ t\ s) = \mathit{numadd}\ (\mathit{simpnum}\ t)\ (\mathit{simpnum}\ s)$
$\mathit{simpnum}\ (\mathit{Sub}\ t\ s) = \mathit{numsub}\ (\mathit{simpnum}\ t)\ (\mathit{simpnum}\ s)$
$\mathit{simpnum}\ (\mathit{Mul}\ i\ t) = (\textsf{if}\ i = 0\ \textsf{then}\ C\ 0\ \textsf{else}\ \mathit{nummul}\ (\mathit{simpnum}\ t)\ i)$
$\mathit{simpnum}\ (\mathit{Floor}\ t) = \mathit{numfloor}\ (\mathit{simpnum}\ t)$
$\mathit{simpnum}\ (\mathit{CN}\ n\ c\ t) = (\textsf{if}\ c = 0\ \textsf{then}\ \mathit{simpnum}\ t\ \textsf{else}\ \mathit{CN}\ n\ c\ (\mathit{simpnum}\ t))$
$\mathit{simpnum}\ (\mathit{CF}\ c\ t\ s) = \mathit{simpnum}\ (\mathit{Add}\ (\mathit{Mul}\ c\ (\mathit{Floor}\ t))\ s)$

Now consider the last equation: Here obviously the argument gets larger, at least if we use the usual size measure, which just counts constructors. However this increase is just temporary, and after a few more evaluation steps, we will end up again with an argument of smaller size. Put differently, if we unfolded the recursive call in the last equation a few times, we would get

$\mathit{simpnum}\ (\mathit{CF}\ c\ t\ s) =$
$\mathit{numadd}\ (\textsf{if}\ c = 0\ \textsf{then}\ C\ 0\ \textsf{else}\ \mathit{nummul}\ (\mathit{numfloor}\ (\mathit{simpnum}\ t))\ c)\ (\mathit{simpnum}\ s)$

where all calls are again structurally decreasing.

In rewriting, this simply corresponds to the rewriting of the right hand sides of the dependency pairs [8, Thm. 21]. The same idea has also been used in an extended termination checker [11] for ACL2 [9], where it is called *merging*, but only justified metatheoretically.

In our framework, we can give a formal justification of this step, using relation composition: Lemma 2 allows us to replace some call R in a goal $\mathit{wf}\ (R \cup S)$ by $(R \cup S) \circ R$ Informally, this means that we compose the R-step with all its possible successors. Since the composition of two relation comprehensions again has the form of a relation comprehension, we again obtain a subgoal of the form that can be handled by the methods described above.

In the *simpnum* example, R would be the problematic call, and S the union of the other calls. After the merging, most resulting calls can be simplified away, since after the *CN* call there must follow an *Add* call and so on. We have to merge the resulting call two more times until the “temporary increase” has gone and the termination proof has become trivial.

7 Conclusion

We have described how dependency pair proofs can be performed in a relational setting. Our approach is more appropriate for termination proofs about functions that are defined by wellfounded recursion in an interactive theorem prover.

The transformations described here are easy to automate in the form of tactics, and we have already implemented some of them. Of course, finding suitable orderings for the simplified problems is still difficult, but heuristics based on measure functions can be applied. In fact, our approach naturally subsumes the existing automation based on lexicographic combinations [3], which can be seen as a repeated application of reduction pairs, but without using the dependency graph.

Perhaps more interesting is that we gave a very simple and non-technical interpretation of (some aspects of) the dependency pair method outside the framework of term rewriting, but in terms of general lemmas about wellfoundedness, union and composition. Compared to term rewriting, the notions in our framework are often slightly more abstract and, in a sense, less operational: For instance, the $\rightarrow^*_{\mathcal{R}}$ relation in the definition of a chain has been replaced by just equality, and the requirement that reduction pairs should be monotonic [8] is dropped altogether. A more detailed analysis of the differences between our approach and traditional dependency pairs is the subject of future work.

Acknowledgement I thank Pierre Courtieu, Cathrine Dubois, and Xavier Urbain for inviting me to Paris and for interesting discussions that stimulated the ideas presented here. Amine Chaieb and Christian Sternagel gave valuable feedback on a draft of this paper.

References

1. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.
2. F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In A. Geser and H. Søndergaard, editors, *Eighth International Workshop on Termination, WST'06, Seattle, WA, USA*, 2006.
3. L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2007*, volume 4732, pages 38–53, 2007.
4. A. Chaieb. Verifying mixed real-integer quantifier elimination. In Furbach and Shankar [6], pages 528–540.
5. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In B. Konev and F. Wolter, editors, *6th International Symposium on Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, Sept. 2007. Springer.
6. U. Furbach and N. Shankar, editors. *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, LNAI 4130. Springer, 2006.
7. J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated termination analysis for Haskell: From term rewriting to programming languages. In F. Pfenning, editor, *RTA*, volume 4098 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2006.
8. J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In F. Baader and A. Voronkov, editors, *LPAR*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2004.
9. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

10. A. Krauss. Partial recursive functions in higher-order logic. In Furbach and Shankar [6], pages 589–603.
11. P. Manolios and D. Vroon. Termination analysis with calling context graphs. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2006.
12. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
13. K. Slind. *Reasoning About Terminating Functional Programs*. PhD thesis, Institut für Informatik, TU München, 1999.