Encoding Induction in Correctness Proofs of Program Transformations as a Termination Problem*

Conrad Rau[1], David Sabel[1], and Manfred Schmidt-Schauß[1]

1 **Dept. Informatik und Mathematik, Inst. Informatik, Goethe-University D-60054 Frankfurt, Germany**
{rau,sabel,schauss}@ki.informatik.uni-frankfurt.de

Abstract

The diagram-based method to prove correctness of program transformations consists of computing complete set of (forking and commuting) diagrams, acting on sequences of standard reductions and program transformations. In many cases, the only missing step for proving correctness of a program transformation is to show the termination of the rearrangement of the sequences. Therefore we encode complete sets of diagrams as term rewriting systems and use an automated tool to show termination, which provides a further step in the automation of the inductive step in correctness proofs.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages

Keywords and phrases Termination, Program Transformations, Correctness

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

The motivation for this work is derived from proving correctness of program transformations in program calculi, in particular in extended lambda calculi that model core-languages of variants of Haskell.

In our setting a *program calculus* is a tuple $(\mathcal{E}, \mathcal{C}, \xrightarrow{sr}, \mathcal{A})$ where $\mathcal{E}$ is the set of *expressions*, $\mathcal{C}$ is the set of *contexts*, i.e. usually $\mathcal{C}$ consists of all expressions of $\mathcal{E}$ where one subexpression is replaced by the context hole, $\xrightarrow{sr} \subseteq \mathcal{E} \times \mathcal{E}$ is a small-step reduction relation (called **s**tandard **r**eduction) which defines the operational semantics of the program calculus and $\mathcal{A} \subseteq \mathcal{E}$ is a set of *answers*, which are usually $\xrightarrow{sr}$-irreducible. The evaluation of a program expression $e \in \mathcal{E}$ is a sequence of standard reduction steps to an answer $a \in \mathcal{A}$, i.e. $e \xrightarrow{sr,*} a$, where $\xrightarrow{sr,*}$ denotes the reflexive-transitive closure of $\xrightarrow{sr}$. If such an evaluation exists, then we write $e{\Downarrow}$ and say e *converges*, otherwise we write $e{\Uparrow}$ and say e *diverges*. The semantics is the *contextual equivalence* of expressions: $e \sim_c e' :\iff e \leq_c e' \wedge e' \leq_c e$, where $e \leq_c e' :\iff \forall C \in \mathcal{C} : C[e]{\Downarrow} \implies C[e']{\Downarrow}$.

A program transformation $\xrightarrow{T} \subseteq (\mathcal{E} \times \mathcal{E})$ is a binary relation on expressions. It is called *correct* if for all e, e' with $e \xrightarrow{T} e'$ the equivalence $e \sim_c e'$ holds. Usually a *context-closure* T' of the program transformation T is considered (w.r.t. all contexts, or a restricted class of contexts, if a context lemma is available), such that proving $e \xrightarrow{T'} e'$ implies $e{\Downarrow} \iff e'{\Downarrow}$ suffices to

* This work was supported by the DFG under grant SCHM 986/9-1.

conclude that T is a correct program transformation. In the following we do not distinguish between T and its context-closure T', and assume that a program transformation is always closed by an appropriate class of contexts such that the correctness proof is reduced to show equivalence of convergence for all $e \xrightarrow{T} e'$. Moreover, with $\xleftarrow{T}$ denoting the inverse of $\xrightarrow{T}$ the correctness of $\xrightarrow{T}$ holds, if $\xrightarrow{T}$ as well as $\xleftarrow{T}$ are convergence preserving (where $\xrightarrow{T}$ is convergence preserving if $e \xrightarrow{T} e' \implies (e{\Downarrow} \implies e'{\Downarrow})$).

In the application below, we will use different program transformations $T_1, \ldots, T_k$ in which case we set $T = \bigcup_{i=1}^{k} T_i$. In general, there are also different kinds of standard reductions, which are used in the concrete proofs, hence we extend the standard reduction to $\xrightarrow{sr} \subseteq \mathcal{E} \times \mathcal{E} \times L$ where L is a set of labels. Sometimes we indicate the label l by writing $\xrightarrow{sr,l}$. We say the program transformation $\xrightarrow{T}$ is *answer-preserving*, if $a \in \mathcal{A}$ and $a \xrightarrow{T} e$ implies $e \in \mathcal{A}$; and *weakly answer-preserving*, if $a \in \mathcal{A}$ and $a \xrightarrow{T} e$ implies $e{\Downarrow}$.

The diagram-based proof method operates on abstract reduction sequences (ARS), which are strings consisting only of the standard reductions with their labels, and the program transformations, but the expressions are ignored with the exception of an abstract symbol A for an answer. A *forking diagram* is a rewriting rule $L \rightsquigarrow R$ on ARSs. The semantics of a diagram $L \rightsquigarrow R$ is that the reduction sequence L can be transformed (or rewritten) into the reduction sequence R. We also allow diagrams that speak about transitive closures of reductions. We are only interested in ARSs that are a mix of $\xleftarrow{sr}$ and $\xrightarrow{T}$-reductions, perhaps labeled, together with an answer token A to the left. The idea of the diagrams is that they transform reduction sequences into evaluations. In general, this rewriting is non-deterministic, which is the price for abstracting away the term structure. *Completeness* of a set $DF(\xrightarrow{T})$ of forking diagrams for transformation $\xrightarrow{T}$ means that every ARS $A \xleftarrow{sr,+} \xrightarrow{T}$ is modifiable by a diagram. For $\xleftarrow{T}$ we call the diagrams in $DF(\xleftarrow{T})$ *commuting diagrams*.

Usually, forking diagrams are of the form $\xleftarrow{sr,l_n} \ldots \xleftarrow{sr,l_1} \xrightarrow{T_k} \rightsquigarrow \xrightarrow{T_1} \ldots \xrightarrow{T_m} \xleftarrow{sr,l_{n'}} \ldots \xleftarrow{sr,l_1}$ where labels l_i may also be omitted and where also the meta-symbols $+$ and $*$ may occur for the transitive/transitive-reflexive closure of a standard reduction or transformation.

We also need another form of diagrams, the *answer diagrams*, $DA(\xrightarrow{T})$, which are called *complete* (for transformation $\xrightarrow{T}$), if every ARS $A \xrightarrow{T,+}$ is modifiable by a diagram in DA. In the case of answer-preservation, these extra diagrams are simply $A \xrightarrow{T} \rightsquigarrow A$, and for a weakly answer-preserving transformation, the diagrams are of the form $A \xrightarrow{T} \rightsquigarrow A \xleftarrow{sr,l_n} \ldots \xleftarrow{sr,l_1}$, where usually only a subset of the labels occur as l_i which may ease the termination proof.

In applications to calculi, the computation of the diagram sets for a given program transformation is done by analyzing the syntax of expressions and the syntax of rules and by covering all possibilities, where usually labels are heavily used at $\xleftarrow{sr}$, depending on the kind of reduction rules, and often, several program transformations occur in the diagram set. This computation may be done by hand, but there is also a proposal for automating this in an expressive core calculus of Haskell, see [3, 4].

The diagram based method to show correctness of a program transformation $\xrightarrow{T}$ is performed by the steps:

1. Show (weak) answer-preservation of $\xrightarrow{T}$ and compute the DA-diagrams.
2. Compute complete sets of forking-diagrams for $\xrightarrow{T}$.
3. Show that every reduction sequence $a \xleftarrow{sr,*} e \xrightarrow{T} e'$ where $a \in \mathcal{A}$ can be transformed using the diagrams from steps 1 and 2 into $a' \xleftarrow{sr,*} e'$, where $a' \in \mathcal{A}$. This is usually done by an induction on the application of diagrams.

4. Do the same by performing steps (1), (2), (3) for the inverse relation $\xleftarrow{T}$.

Since answer-preservation implies weak answer-preservation, we show the next theorem only for the weak case.

▶ Proposition 1.1. Let $\xrightarrow{T}$ be weakly answer preserving, and let $DF(\xrightarrow{T})$ and $DA(\xrightarrow{T})$ be the complete sets of forking and answer diagrams, respectively, for $\xrightarrow{T}$. Then termination of $DF(\xrightarrow{T}) \cup DA(\xrightarrow{T})$ implies that $\xrightarrow{T} \subseteq \leq_c$.

Proof. Starting with $e{\Downarrow}$ and $e \xrightarrow{T} e'$, this corresponds to an ARS of the form $A \xleftarrow{sr,l_n} \ldots \xleftarrow{sr,l_1} \xrightarrow{T}$. Completeness of $DF(\xrightarrow{T})$ and $DA(\xrightarrow{T})$ guarantees that an ARS in normal-form is of the form $A \xleftarrow{sr,l'_m} \ldots \xleftarrow{sr,l'_1}$, which shows $e'{\Downarrow}$. Since $\xrightarrow{T}$ is assumed to be closed for context application, this implies $e \leq_c e'$. Since this holds for all $e \xrightarrow{T} e'$, we have shown $\xrightarrow{T} \subseteq \leq_c$. ◀

▶ **Theorem 1.2.** *If the assumptions of Proposition 1.1 hold for $\xrightarrow{T}$ as well as for $\xleftarrow{T}$ – including complete sets $DF(\xrightarrow{T}), DF(\xleftarrow{T}), DA(\xrightarrow{T})$, and $DA(\xleftarrow{T})$ – then $\xrightarrow{T} \subseteq \sim_c$, which means that T is a correct program transformation.*

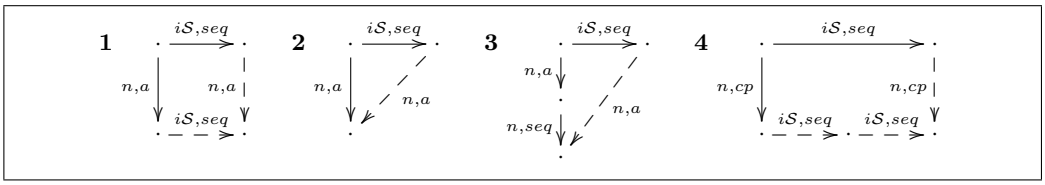
Based on the description above and Theorem 1.2, we encode reduction sequences as terms, and complete sets of diagrams as term rewriting systems on the sequences. As we will demonstrate for encoding some of our diagrams including transitive closure we require *conditional integer term rewriting systems* (ITRS). However, these can also be treated by the automated termination prover AProVE [1, 2]. Hence we can use the AProVE system to show termination of TRSs / conditional ITRSs, which provides a further step in the automation of the inductive step in correctness proofs.

2 Encodings of Reductions and Sets of Diagrams

We give some examples for the encoding of complete sets of diagrams into (I)TRSs. The diagrams are taken from [5] for an extended call-by-need lambda calculus with a standard reduction called *normal order reduction*, denoted as $\xrightarrow{n}$, and expressions considered as answers are called *weak head normal forms* (WHNFs).

We first consider the transformation *seq*. Figure 1a shows the forking diagrams $DF(\xrightarrow{i\mathcal{S},seq})$ for the transformation $(i\mathcal{S}, seq)$, which is the context-closure of *seq*. The label a signifies an arbitrary reduction label. The solid lines in the diagrams represent the left hand sides and the dashed lines the right hand sides in the diagram rules of the form $L \rightsquigarrow R$. The diagrams can also be represented in their flat form, e.g. the flat form of the first diagram is $\xleftarrow{n,a} \xrightarrow{i\mathcal{S},seq} \rightsquigarrow \xrightarrow{i\mathcal{S},seq} \xleftarrow{n,a}$. Figure 1b shows the TRS encoding of the forking- and answer-diagrams for the transformation $(i\mathcal{S}, seq)$, where x is a variable, and all other symbols are function symbols. We highlight on some properties of the encoding:

- The special answer token (i.e. WHNF-token) is represented as the constant w.
- The abstract reduction sequences in the graphical diagrams are encoded from *right to left*, i.e. the flat diagram $\xleftarrow{n,a} \xrightarrow{i\mathcal{S},seq} \rightsquigarrow \xrightarrow{i\mathcal{S},seq} \xleftarrow{n,a}$ is represented as the rewrite rule $iSseq(n(a,x)) \to n(a, iSseq(x))$. This is done to express the fact that diagrams turn reduction sequences into evaluations. E.g. the sequence $w \xleftarrow{n,a} \xleftarrow{n,a} \xleftarrow{n,a} \xrightarrow{i\mathcal{S},seq}$ is represented by the term $iSseq(n(a,n(a,n(a,w))))$ and can be turned into the evaluation $w \xleftarrow{n,a} \xleftarrow{n,a} \xleftarrow{n,a}$ (either by repeated application of the first diagram and a closing application of the answer-diagram or by a single application of the second diagram).



(a) Forking diagrams for the transformation $(i\mathcal{S}, seq)$

	$iSseq(n(a,x))$	$\to$	$n(a, iSseq(x))$
1	$iSseq(n(seq,x))$	$\to$	$n(seq, iSseq(x))$
	$iSseq(n(cp,x))$	$\to$	$n(cp, iSseq(x))$
	$iSseq(n(a,x))$	$\to$	$n(a,x)$
2	$iSseq(n(seq,x))$	$\to$	$n(seq,x)$
	$iSseq(n(cp,x))$	$\to$	$n(cp,x)$
	$iSseq(n(a,n(seq,x)))$	$\to$	$n(a,x)$
3	$iSseq(n(seq,n(seq,x)))$	$\to$	$n(seq,x)$
	$iSseq(n(cp,n(seq,x)))$	$\to$	$n(cp,x)$
4	$iSseq(n(cp,x))$	$\to$	$n(cp, iSseq(iSseq(x)))$
Answer diagram	$iSseq(w)$	$\to$	w

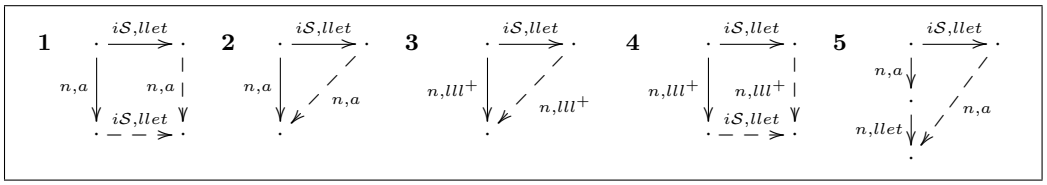
(b) TRS encoding of forking- and answer-diagrams for the transformation $(i\mathcal{S}, seq)$

Figure 1 Diagrams and their TRS encoding for the transformation $(i\mathcal{S}, seq)$

- The labels of normal order reductions and transformations are encoded differently: Labels of transformations are encoded directly into function symbols (like *iSseq*) whereas labels of normal order reductions are encoded as parameters of function applications, e.g. in the term $n(a,x)$ the constant a denotes the label of the normal order reduction. Here a is a constant that represents arbitrary reduction labels (that are not *seq* or *cp*) whereas the constants *seq* and *cp* denote those specific labels (this is also the reason why we need three rewrite rules per diagram in the present example). The different encoding of names has mainly technical reasons: The automatic proofs using AProVE are in some cases only possible with the described encoding.

Since *seq* is answer-preserving the TRS encoding of $DA(\xrightarrow{i\mathcal{S},seq})$ consists of the single diagram $iSseq(w) \to w$. For the *seq* transformation the termination of the TRS encoded complete diagram set could be automatically shown.

Figure 2a gives another example of a complete set of forking diagrams $DF(\xrightarrow{i\mathcal{S},llet})$ for the transformation $(i\mathcal{S}, llet)$, which is answer-preserving. In the diagrams $\xrightarrow{n,lll^+}$ represents a (non-empty) sequence of *lll*-reductions i.e. the transitive closure of those reductions. These symbols require a special treatment in the encoding into TRS, since they represent an infinite set of diagrams. If the symbol $\xrightarrow{n,lll^+}$ occurs on the left hand side of a diagram, this means that any given (non-empty) reduction sequence of *lll*-reductions can be matched. In the encoding this symbol is represented by the function symbol *nlllPlusL* and there are additional rules which allow to contract a given sequence of *lll*-reductions into the symbol $\xrightarrow{n,lll^+}$ (see Figure 2c). If a symbol $\xrightarrow{n,lll^+}$ occurs on the right hand side of a diagram, then a naive approach would be to add rules $\xrightarrow{n,lll^+} \rightsquigarrow \xrightarrow{n,lll}$ and $\xrightarrow{n,lll^+} \rightsquigarrow \xrightarrow{n,lll^+}\xrightarrow{n,lll}$. However, this approach does not work, since it introduces nontermination in the corresponding TRS. Hence, we use integer term rewrite systems for the encoding, which allow to rewrite the symbol $\xrightarrow{n,lll^+}$ into a sequence of $\xrightarrow{n,lll}$-reductions of arbitrary but *fixed* length. In the encoding we use the function symbol *nlllPlusR* for the occurrence of $\xrightarrow{n,lll^+}$ on the right hand side. For diagrams 3 and 4, an integer variable k is introduced by the rewriting rule which is like guessing a natural number. Additionally we add ITRS-rules to rewrite the symbol into a sequence of k $\xrightarrow{n,lll}$-reductions (see Figure 2d).



(a) Forking diagrams for the transformation $(i\mathcal{S}, llet)$

3 $iSllet(nlllPlusL(x)) \to nlllPlusR(k,x)$	**4** $iSllet(nlllPlusL(x)) \to nlllPlusR(k, iSllet(x))$

(b) ITRS encoding of the third and fourth forking diagram for the transformation $(i\mathcal{S}, llet)$

$n(lll, nlllPlusL(x)) \to nlllPlusL(x)$ $n(lll, x) \to nlllPlusL(x)$	$nlllPlusR(0,x) \to x$ $nlllPlusR(k,x) \to nlllPlusR(k-1, n(lll,x))$ if $k > 0$

(c) Contracting $\xrightarrow{n,lll}$-sequences into $\xrightarrow{n,lll^+}$ **(d)** Expansion of $\xrightarrow{n,lll^+}$ into k $\xrightarrow{n,lll}$-reductions

Figure 2 Diagrams and ITRS encoding for the transformation $(i\mathcal{S}, llet)$

Termination of $DF(\xrightarrow{i\mathcal{S},llet}) \cup DA(\xrightarrow{i\mathcal{S},llet})$ can be automatically checked using AProVE.

Conclusion We tested the complete sets of (forking as well as commuting) diagrams of several program transformations from [5] and they could all be shown terminating with the above method using AProVE as a tool for automatic termination proofs. While the encoding of most of the diagrams from [5] was in general rather straightforward, there are also cases, where additional knowledge (beyond the mere information of the diagram) has to be employed in the encoding, or where the automatic proof can only be found for a particular syntactic variant. An increasing set of (I)TRS-encoded diagrams and the corresponding termination proofs in AProVE can be found on the website:
`http://www.ki.informatik.uni-frankfurt.de/research/dfg-diagram/auto-induct/`.

Future work is to connect the automated termination prover with the diagram calculator of [3, 4] and thus to complete the tool for automated correctness proofs of program transformations. Another direction is to check more sets of diagrams which probably requires to develop more sophisticated encoding techniques.

Acknowledgements We thank Carsten Fuhs for pointing us to ITRSs for the encoding of transitive closures, and for his support on the AProVE tool. We also thank the anonymous reviewers for their valuable comments on this paper.

References

1 AProVE website, 2011. `http://aprove.informatik.rwth-aachen.de`.
2 C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *20th RTA*, *LNCS* 5595, pp. 32–47. Springer, 2009.
3 C. Rau and M. Schmidt-Schauß. Towards correctness of program transformations through unification and critical pair computation. In *24th UNIF*, *EPTCS* 42, pp. 39–54, 2010.
4 C. Rau and M. Schmidt-Schauß. A unification algorithm to compute overlaps in a call-by-need lambda-calculus with variable-binding chains. In *25th UNIF*, pp. 35–41, 2011.
5 M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.