Automating the Diagram Method to Prove Correctness of Program Transformations

David Sabel
Goethe-University
Frankfurt am Main, Germany
sabel@ki.informatik.uni-frankfurt.de *

We report on the automation of a technique to prove the correctness of program transformations in higher-order program calculi which may permit recursive let-bindings as they occur in functional programming languages. A program transformation is correct if it preserves the observational semantics of programs. In our LRSX Tool the so-called diagram method is automated by combining unification, matching, and reasoning on alpha-renamings on the higher-order meta-language, and automating induction proofs via an encoding into termination problems of term rewrite systems. We explain the techniques, we illustrate the usage of the tool, and we report on experiments.

1 Introduction

Program transformations replace program fragments by program fragments. They are applied as optimizations in compilers, in code refactoring to increase maintainability of the source code, and in verification for equational reasoning on programs. In all cases correctness of the transformations is an indispensable requirement. We focus on program calculi with a small-step operational semantics (in form of a reduction semantics with evaluation contexts, see e.g. [23]) and a notion of successfully evaluated programs. Convergence of programs holds, if the program can be evaluated to a successful program. As program equivalence we use contextual equivalence [9, 10], which holds for program fragments P_1 and P_2 if interchanging P_1 by P_2 in any program (i.e. context) is not observable w.r.t. convergence. We are particularly interested in extended lambda-calculi with call-by-need evaluation modeling the (untyped) core languages of lazy functional programming languages like Haskell (see [2, 1, 20]).

The LRSX Tool[1] supports correctness proofs of program transformations in those calculi by automating the "diagram method" (see e.g. [20, 15] and also [7, 22]) which was used in earlier work in non-automated pen-and-paper proofs. The diagram method is a syntactic approach that can roughly be outlined as follows. First all overlaps between standard reduction steps and transformation steps are computed, then the overlaps have to be joined resulting in a complete set of diagrams. This step is related to computing and joining critical pairs in term rewrite systems (see e.g. [3]), however, with two rewrite relations and where for one rewrite relation a strategy (defined by the standard reduction) has to be respected. Finally, the diagrams are used in an inductive proof to show correctness of the transformation.

The automation of the method is schematically depicted in Fig. 1. The input consists of a calculus description and a set of program transformations. First the diagram calculator computes the overlaps and then tries to join them. If a complete set of diagrams is obtained, it is translated into a term rewrite system such that termination of the system implies correctness of the program transformations. The automated termination prover AProVE [5] and the certifier CeTA [21] are used to automate these steps.

*This research is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA2908/3-1

[1] available from http://goethe.link/LRSXTOOL61

J. Niehren & D. Sabel (eds.): Rewriting Techniques
for Program Transformations and Evaluation (WPTE 2018)
EPTCS 289, 2019, pp. 17–33, doi:10.4204/EPTCS.289.2

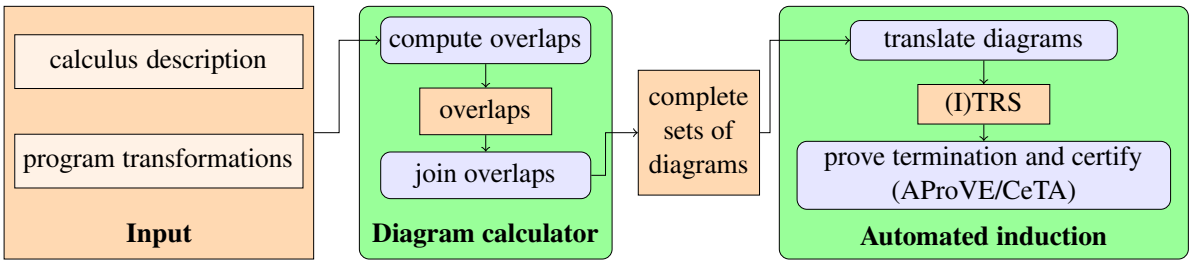


Figure 1: The overall structure of the automated diagram method

Expressions	e	$::=$	$\bot \mid \top \mid (\neg e) \mid (e \wedge e)$	Standard reduction $\xrightarrow{sr}$	
Evaluation contexts	A	$::=$	$[\cdot] \mid \neg A \mid A \wedge e$	(sr, bot)	$A[\bot \wedge e] \to A[\bot]$
General contexts	C	$::=$	$[\cdot] \mid \neg C \mid C \wedge e \mid e \wedge C$	(sr, top)	$A[\top \wedge e] \to A[e]$
Successful programs	$\top$			$(sr, neg, 1)$	$A[\neg\top] \to A[\bot]$
Program transformation	(top)		$C[\top \wedge e] \to C[e]$	$(sr, neg, 2)$	$A[\neg\bot] \to A[\top]$

Figure 2: Syntax and Operational Semantics of the Calculus *Simple*

In previous work, we published results on core algorithms that are used in the tool. In [18] the underlying unification-algorithm was defined and analysed, in [13, 14] a matching algorithm was developed, in [12] a procedure to alpha-rename meta-expressions was presented, and in the work [11] the encoding of the diagrams as term rewrite systems for automating the induction step was developed. However, none of these works presents the full automation of the method. Thus, in this paper, we explain core components of the automated method and illustrate the use of the LRSX Tool. In particular, we provide a formal formulation of the rewrite rules (together with some conditions) which ensure that i) the diagram method is correct and ii) the previously developed algorithms are applicable. We also illustrate how the syntax and rules of a correctness problem are represented in our tool.

Outline. In Sect. 2 we illustrate the diagram method for a simple example and thereafter briefly recall the call-by-need lambda calculus L_{need} which will be our running example throughout the paper. In Sect. 3 we explain the meta language and the representation of the input of the diagram method. In Sect. 4 we describe the automated correctness proof for the standard cases. In Sect. 5 we discuss extensions of the automated correctness proof which are also built in the tool. Also cases which cannot be handled by the tool are discussed. In Sect. 6 we report on some experiments. We conclude in Sect. 7.

2 Illustration of the Diagram Method – Examples

We illustrate the concept of observational semantics, correctness of program transformations, and the diagram method (and its automation) using a quite simple example. In Fig. 2 we define a program calculus *Simple*. The syntax of *Simple*-expressions consists of two constants, $\bot$ to represent a failing computation, and $\top$ to represent success, a unary operator $\neg$ for negation, and a binary operator $\wedge$ which computes the conjunction of $\top$ and $\bot$, i.e. evaluation of $e_1 \wedge e_2$ results in $\top$ iff e_1 and e_2 both evaluate to $\top$ and otherwise the evaluation ends with $\bot$. The reduction strategy which evaluates the $\wedge$-operator from left to right is defined by using evaluation contexts A (defined in Fig. 2 where $[\cdot]$ denotes the context hole). The standard reduction $\xrightarrow{sr}$ is the union of the rules (sr, bot), (sr, top), $(sr, neg1)$, and $(sr, neg2)$.

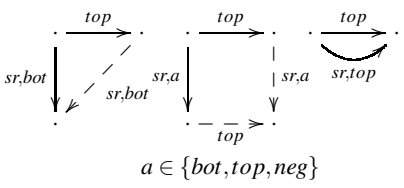


Figure 3: Forking diagrams for (top)

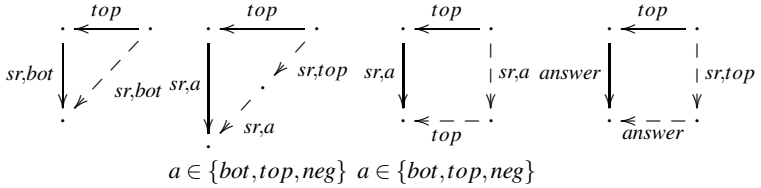


Figure 4: Commuting diagrams for (top)

Evaluation contexts A uniquely determine the position where the next standard reduction has to be applied. Hence, thus standard reduction is deterministic. Let $\xrightarrow{sr,*}$ denote the reflexive-transitive closure of $\xrightarrow{sr}$. A *Simple*-expression e converges (written $e\downarrow$) iff it evaluates to $\top$, i.e. $e\downarrow$ iff $e \xrightarrow{sr,*} \top$. For instance, $((\neg\bot) \wedge \top) \wedge (\neg(\top \wedge \bot)) \xrightarrow{sr} (\top \wedge \top) \wedge (\neg(\top \wedge \bot)) \xrightarrow{sr} \top \wedge (\neg(\top \wedge \bot)) \xrightarrow{sr} \neg(\top \wedge \bot) \xrightarrow{sr} \neg\bot \xrightarrow{sr} \top$.

With C we denote arbitrary contexts. Expressions e_1, e_2 are contextually equivalent, written $e_1 \sim_c e_2$ iff $\forall C : C[e_1]\downarrow \iff C[e_2]\downarrow$. A program transformation P is a binary relation on *Simple*-expressions and it is correct if for all $(e_1, e_2) \in P$, $e_1 \sim_c e_2$ holds.

We consider the correctness proof of transformation (top) which is defined in Fig. 2. Since transformation (top) is already closed by all contexts (i.e. $e_1 \xrightarrow{top} e_2$ implies $C[e_1] \xrightarrow{top} C[e_2]$ for all contexts C), it suffices to show "convergence equivalence" of the transformation, i.e.:

$$\text{(1) for all } e_1 \xrightarrow{top} e_2\text{: } e_1\downarrow \implies e_2\downarrow \text{ and (2) for all } e_1 \xrightarrow{top} e_2\text{: } e_2\downarrow \implies e_1\downarrow.$$

For part (1), we have to find all the cases where $e_1\downarrow$ and $e_1 \xrightarrow{top} e_2$. A first case distinction is whether (i) e_1 is already successful (i.e. $e_1 = \top$) or (ii) e_1 is reducible by the standard reduction. To systematically compute a finite representation of all cases for e_1 and e_2, we use unification and thus unify all left hand sides of rule (top) with $\top$ (for case (i)) and also with all left hand sides of all standard reductions (for case (ii)). Let us consider one of those unifications: we unify the left hand sides of (top) and (sr, bot). The unification problem consists of the equation $C[\top \wedge S_1] \doteq A[\bot \wedge S_2]$, where C and A are meta-variables for C- and A-contexts and S_1, S_2 are meta-variables for *Simple*-expressions. It has two most general unifiers: either $(\top \wedge S_1)$ is a subexpression of S_2, or $(\top \wedge S_1)$ and $(\bot \wedge S_2)$ are at parallel positions.

We only illustrate the former case. The unifier is $\sigma = \{S_2 \mapsto C_1[\top \cap S_1], C \mapsto A[\bot \cap C_1]\}$ and the instantiated expression is $\sigma(C[\top \wedge S_1]) = A[\bot \cap C_1[\top \wedge S_1]] = \sigma(A[\bot \wedge S_2])$. After instantiating the right hand sides of the rules with the unfier, we get $\sigma(C[S_1]) = A[\bot \cap C_1[S_1]]$ and $\sigma(A[\bot]) = A[\bot]$. The sequence $A[\bot] \xleftarrow{sr,bot} A[\bot \cap C_1[\top \wedge S_1]] \xrightarrow{top} A[\bot \cap C_1[S_1]]$ is called a forking overlap. It has to be joined by applying standard reductions for the right and transformation steps for the left meta-expression to find a common successor of both. If we apply a standard reduction to $A[\bot \cap C_1[S_1]]$, i.e. $A[\bot \cap C_1[S_1]] \xrightarrow{sr} A[\bot]$, we have already found a join. Note that this "application" of rules is done on meta-expressions which contain meta-variables for contexts and expressions and thus it can be done by matching the expressions against the left hand side of the transformation or standard reduction, resp.

The fork together with its join is called a forking diagram. Usually, forking diagrams are represented abstractly by removing the concrete expressions. Computing all unifiers and joins leads to the set of (abstract) diagrams shown in Fig. 3. These diagrams can be used in an inductive proof to show that if $e_1 \xrightarrow{top} e_2$ then $e_1\downarrow \implies e_2\downarrow$. We use induction on the length of the reduction sequence from e_1 to $\top$. If e_1 is successful, then the claim holds. For the induction step, let $e_1 \xrightarrow{sr} e_1'$ such that $e_1'\downarrow$. Applying a diagram to the fork $e_1' \xleftarrow{sr} e_1 \xrightarrow{top} e_2$ either shows that $e_1' = e_2$, or that there exists e_2' with $e_2 \xrightarrow{sr} e_2'$ and either $e_2' = e_1'$ or $e_1' \xrightarrow{top} e_2'$. The induction hypothesis applied to e_1' shows that $e_2'\downarrow$ and thus $e_2\downarrow$.

Expressions e and environments Env where v, v_i, w, w_i are variables,

$e ::= w \mid \lambda w.e \mid (e_1\ e_2) \mid \texttt{letrec}\ Env\ \texttt{in}\ e \qquad Env ::= w_1{=}e_1, \ldots, w_n{=}e_n$

Application contexts A and reduction contexts R

$A ::= [\cdot] \mid (A\ e) \qquad R ::= A \mid \texttt{letrec}\ Env\ \texttt{in}\ A \mid \texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}A_m, Env\ \texttt{in}\ A_0[w_1]$

Standard reduction $\xrightarrow{sr}$

(sr,lbeta) $R[((\lambda w.e_1)\ e_2)] \to R[\texttt{letrec}\ w{=}e_2\ \texttt{in}\ e_1]$

(sr,lapp) $R[(\texttt{letrec}\ Env\ \texttt{in}\ e_1)\ e_2] \to R[\texttt{letrec}\ Env\ \texttt{in}\ (e_1\ e_2)]$

(sr,cp-in) $\texttt{letrec}\ \{w_i{=}w_{i+1}\}_{i=1}^{m-1}, w_m{=}\lambda w.e, Env\ \texttt{in}\ A_0[w_1]$
$\to \texttt{letrec}\ \{w_i{=}w_{i+1}\}_{i=1}^{m-1}, w_m{=}\lambda w.e, Env\ \texttt{in}\ A_0[\lambda w.e]$

(sr,cp-e) $\texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}A_m[v_1], \{v_j{=}v_{j+1}\}_{j=1}^{n-1}, v_n{=}\lambda w.e, Env\ \texttt{in} A[w_1]$
$\to \texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}A_m[\lambda w.e], \{v_j{=}v_{j+1}\}_{j=1}^{n-1}, v_n{=}\lambda w.e, Env\ \texttt{in} A[w_1]$
where $A_m{\neq}[\cdot], m{\geq}1, n{\geq}1$

(sr,llet-in) $\texttt{letrec}\ Env_1\ \texttt{in}\ \texttt{letrec}\ Env_2\ \texttt{in}\ e \to \texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e$

(sr,llet-e) $\texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}(\texttt{letrec}\ Env_1\ \texttt{in}\ e), Env_2\ \texttt{in}\ A_0[w_1]$
$\to \texttt{letrec}\ \{w_i{=}A_i[w_{i+1}]\}_{i=1}^{m-1}, w_m{=}e, Env_1, Env_2\ \texttt{in}\ A_0[w_1]$

Successful programs are $\lambda w.e$ or $\texttt{letrec}\ Env\ \texttt{in}\ \lambda w.e$ *called weak head normal forms (WHNFs)*

Garbage Collection

(gc1) $\texttt{letrec}\ w_1{=}e_1, \ldots, w_n{=}e_n, Env\ \texttt{in}\ e \to \texttt{letrec}\ Env\ \texttt{in}\ e$, if all w_i do not occur in Env, e

(gc2) $\texttt{letrec}\ w_1{=}e_1, \ldots, w_n{=}e_n\ \texttt{in}\ e \to e$, if all w_i do not occur in e

Copy Transformation

(cp-in) $\texttt{letrec}\ w{=}\lambda v.e, Env\ \texttt{in}\ C[w] \to \texttt{letrec}\ w{=}\lambda v.e, Env\ \texttt{in}\ C[\lambda v.e]$

(cp-e) $\texttt{letrec}\ w_1{=}\lambda v.e, w_2{=}C[w_1], Env\ \texttt{in}\ e' \to \texttt{letrec}\ w_1{=}\lambda v.e, w_2{=}C[\lambda v.e], Env\ \texttt{in}\ e'$

Figure 5: The calculus L_{need}

Part (2) of the correctness proof of transformation (top) is analogous, but we have to overlap (and also unify) the *right* hand side of (top) of against the successful result $\top$ and against any left hand side of a standard reduction. The obtained set of diagrams is shown in Fig. 4. The last diagram is for the case that $e \xrightarrow{top} \top$. Then also $e \xrightarrow{sr,top} \top$. This is expressed by the diagram where we added the rule $\top \xrightarrow{answer} ans$ for a new constant *ans* (representing answers, i.e. successful results).

By induction and using the diagrams we can show that $e_1 \xrightarrow{top} e_2$ and $e_2{\downarrow}$ also implies $e_1{\downarrow}$. This completes the diagram-based correctness proof for (top) and the program calculus *Simple*. As we explain later in Sect. 4, the induction can be automated by interpreting the diagrams as rewrite rules on their sequences of labels (where sequences of solid arrows are replaced by the sequences with dashed arrows). In Appendix B we provide the full input for the LRSX Tool that is necessary to describe the calculus *Simple*, the transformation (top), and to perform the automatized correctness proof of (top).

As illustrated before, the diagram computation can be done by algorithms for unification and matching, where for the *Simple* calculus we require them for first-order terms extended by meta-variables for contexts. For such a language, all these parts can be implemented by known algorithms and techniques (by using some occurrence restrictions on the context variables, also efficiently, while the general problem is known to be in PSPACE [6]). However, we are interested in languages with more complicated syntactic constructs, i.e. program calculi with expressions with binders (i.e. higher-order constructs, like lambda-abstraction) and with recursive bindings (called letrec-expressions). This means, that we have to use an extended meta-language which, for instance, is capable to represent binders and

letrec-expressions. That is why we from now on switch to a more complex running example, the call-by-need lambda calculus with letrec L_{need} (see e.g. [19] for the calculus L_{need} and e.g. [2, 1] for similar calculi). Its syntax, small-step operational semantics (called standard reduction), and the program transformations (gc1) and (gc2) to perform garbage collection, and (cp-in) and (cp-e) to copy abstractions, are shown in Fig. 5. Syntactically, L_{need} extends the untyped lambda calculus by `letrec`-expressions `letrec` $w_1 = e_1, \ldots, w_n = e_n$ `in` e where the `letrec`-environment $w_1 = e_1, \ldots, w_n = e_n$ represents a set of *unordered* bindings which have a recursive scope, i.e. the scope of w_i are all expressions $e_1, \ldots, e_n$ as well as the `in`-expression e. Standard reduction implements the lazy evaluation strategy with sharing by applying small-step reduction rules at needed positions which are determined by application contexts A, reduction contexts R, and chains of `letrec`-bindings that occur as variable-to-variable bindings and also as chains $\{w_i = A[w_{i+1}]\}_{i=1}^{m}$. The rule (sr,lbeta) implements β-reduction with sharing, the rules (sr,lapp), (sr,llet-in), and (sr,llet-e) reorder and join letrec-environments , the rules (sr,cp-in) and (sr,cp-e) copy abstractions into needed positions. Reduction is meant modulo (extended) α-renaming, i.e. α-equivalent expressions where `letrec`-bindings are treated like a set are not distinguished.

3 Representation of Program Calculi and Transformations

The input of the diagram technique is a program calculus – with definitions of contexts, standard reduction rules, answers representing successfully evaluated programs – and a set of program transformations.

3.1 Meta-Syntax to Represent Expressions

We represent rules and answers in the meta-language `LRSX` (see also [18]). To cover several program calculi the representation is parametrized over a set $\mathscr{F}$ of (higher-order) function symbols and a finite set $\overline{K}$ of context classes[2]. A context class describes a set of contexts (usually defined by a grammar), like A- and C-contexts in *Simple* or in L_{need}. We define the *syntax of* `LRSX`*-expressions* **Exp**, the syntax of variables of a countably-infinite set of variables **Var**, the syntax of *higher-order expressions of order n* $\mathbf{HExp}^n$ (i.e. syntactic constructs that bind / abstract over n variables, in particular $\mathbf{HExp}^0 = \mathbf{Exp}$), and the syntax of *environments* **Env** and *bindings* **Bind**. We we assume that every $f \in \mathscr{F}$ has a *syntactic* type of the form $f : \tau_1 \to \ldots \to \tau_{ar(f)} \to \mathbf{Exp}$, where τ_i may be **Var** or $\mathbf{HExp}^{k_i}$, i.e. the syntactic type of f defines the arity of f, but also the syntactic category of which each argument has to be part of. If not otherwise stated, we always assume $\{\mathtt{var}, \lambda\} \subseteq \mathscr{F}$ where function symbol `var` of type $\mathbf{Var} \to \mathbf{Exp}$ lifts variables to expressions, and λ has type $\mathbf{HExp}^1 \to \mathbf{Exp}$. To distinguish term variables, meta-variables, and meta-symbols, we use different fonts and lower- or upper-case letters: concrete term-variables of type **Var** are denoted by x, y, and x, y are used as meta-symbols to denote a concrete term variable or a meta-variable. Similarly, s, t denote expressions, *env* denotes environments, and b denotes bindings. Meta-variables are written in upper-case letters, where X, Y are of type **Var**, S is of type **Exp**, E is of type **Env**, D is a context variable, and *Ch* is a two-hole environment-context variable (chain variable, for short). Each context variable D has a class $cl(D)$ and each *Ch*-variable has a class $cl(Ch)$. The grammars for the different syntactic categories are:

$$x, y, z \in \mathbf{Var} ::= X \mid \mathtt{x}$$
$$s, t \in \mathbf{HExp}^0 ::= S \mid D[s] \mid \mathtt{letrec}\ env\ \mathtt{in}\ s \mid f\, r_1 \ldots r_{ar(f)} \text{ such that } r_i \in \tau_i \text{ if } f : \tau_1 \to \ldots \to \tau_n \to \mathbf{Exp}$$
$$s \in \mathbf{HExp}^n ::= x.s_1 \quad \text{if } s_1 \in \mathbf{HExp}^{n-1} \text{ and } n \geq 1$$
$$b \in \mathbf{Bind} ::= x{=}s \quad \text{where } s \in \mathbf{HExp}^0 \qquad env \in \mathbf{Env} ::= \emptyset \mid E; env \mid Ch[x, s]; env \mid b; env$$

[2] In the LRSX Tool the set $\overline{K}$ has to be defined explicitly while the set $\mathscr{F}$ is extracted from the used symbols in the input.

```
define A ::= [.] | (app A S)
define T ::= [.] | (app T S) | (app S T)
          | letrec X=T;E in S
          | letrec E in T where E /= {}
declare prefix A A = (A,A)
declare prefix A T = (A,T)
declare prefix T A = (A,A)
declare prefix T T = (T,T)
```

```
declare fork A T = (A,A,T,(app [.1] [.2]))
declare fork T T = (T,T,T,(app [.1] [.2]))
declare fork T T = (T,T,T,(app [.2] [.1]))
declare fork T T = (T,T,T,(letrec X=[.1];E in [.2]))
declare fork T T = (T,T,T,(letrec X=[.2];E in [.1]))
declare fork T T =
                    (T,T,T,(letrec X=[.1];Y=[.2];E in S))
declare fork T A = (A,T,A,(app [.2] [.1]))
```

Figure 6: Definition of application and top-contexts as input for the LRSX Tool

An LRSX-expression s is *ground* (written as s) iff it does not contain any meta-variable, d denotes a ground context and d denotes contexts, that may contain meta-variables. Filling the hole of d with s is written as $d[s]$. Multi-contexts with $k > 1$ holes are written with several hole symbols $[\cdot_1], \ldots, [\cdot_k]$.

Example 3.1. The syntax of the calculus *Simple* can be represented by instantiating $\mathscr{F} = \{\bot, \top, \neg\}$ where $\bot, \top :$ **Exp**, $\wedge :$ **Exp** $\to$ **Exp** $\to$ **Exp**, $\neg :$ **Exp** $\to$ **Exp** and using the context classes $\overline{K} := \{A, C\}$ with corresponding descriptions of them (see below). Assuming that D is a context variable of class A, the expression $D[S_1 \wedge S_2]$ describes all ground expressions of the form $\mathsf{d}[\mathsf{s}_1 \wedge \mathsf{s}_2]$ where d is a ground-context of context class A and S_1, S_2 are arbitrary ground expressions of the calculus *Simp*.

Example 3.2. The syntax of the λ-calculus (and also of our running example L_{need}, since letrec is built-in in LRSX) can be expressed in LRSX, by the function symbols $\mathscr{F} = \{\mathtt{var}, \lambda, \mathtt{app}\}$ where app is a binary function symbol of type **Exp** $\to$ **Exp** $\to$ **Exp**. The application of the identity function to itself can be written as the LRSX-expression app (λ(x.var x)) (λ(x.var x)). Lists can be represented by function symbols nil :: **Exp** and cons :: **Exp** $\to$ **Exp** $\to$ **Exp**. A case-expression – usually written as case l of (Nil $\to e_1$) (Cons x xs $\to e_2$) – to deconstruct lists can be represented as caselist l e_1 x.xs.e_2 where caselist is a function symbol of type **Exp** $\to$ **Exp** $\to$ **HExp**^{2} $\to$ **Exp**. For the context classes, we may use $\overline{K} := \{A, T, C\}$ where C are general contexts, T are top-contexts (which do not have the hole inside an abstraction). Reduction contexts R are not necessary since they can be expressed by A-contexts and several variants of the same reduction rule, for the different kinds of R-contexts.

In addition to a description of the syntax (by a grammar that describes a set of contexts), we require for each context class $\mathscr{K} \in \overline{K}$ a *prefix* and a *forking table*. These tables are used in the matching and unification algorithms to proceed with equations of the form $D_1[s_1] \doteq D_2[s_2]$: the *prefix table* is a partial function that maps pairs of classes ($\mathscr{K}_1$, $\mathscr{K}_2$) to a pair of classes ($\mathscr{K}_3$, $\mathscr{K}_4$) such that for context variables D_i with $cl(D_i) = \mathscr{K}_i$ an equation $D_1[s] \doteq D_2[t]$ where D_1 is a prefix of context D_2, can be replaced by the equation $s \doteq D_4[t]$ and the substitution $\{D_1 \mapsto D_3,\ D_2 \mapsto D_3[D_4]\}$. Undefined cases express that the prefix situation is impossible. The *forking table* is a partial function that maps pairs of classes ($\mathscr{K}_1$, $\mathscr{K}_2$) to a set of tuples of the form ($\mathscr{K}_3$, $\mathscr{K}_4$,$\mathscr{K}_5$,$d[\cdot_1, \cdot_2]$) such that for context variables D_i of class $\mathscr{K}_i$ an equation $D_1[s] \doteq D_2[t]$ where the paths to the holes of D_1 and D_2 fork, the equation can be removed by guessing one tuple in the set and substituting $D_1 \mapsto D_3[d[D_4[\cdot], D_5[t]]], D_2 \mapsto D_3[d[D_4[s], D_5[\cdot]]]$.

We do not know whether the prefix and the forking table can be computed from given grammars for the context classes. Thus, in the LRSX Tool, the user has to specify them as part of the input. For calculus *Simple*, the definition of these tables is shown in the Appendix B. For calculus L_{need}, we define classes for application contexts A, top contexts T and arbitrary contexts C. The definition of the former two classes as input for the LRSX Tool is shown in Fig. 6. We illustrate some exemplary entries of the prefix and forking table: the prefix table maps (A, T) to (A, T), since for every application context D_1 that is a prefix of a top-context D_2, we can substitute $D_1 \mapsto D_3$ and $D_2 \mapsto D_3[D_4]$ where D_3 must be an application context (since D_1 is one) and D_4 must be a top context (since D_2 is one). The prefix table

maps (T,A) to (A,A), since for every top-context D_1 that is a prefix of an application context D_2, we can substitute $D_1 \mapsto D_3$ and $D_2 \mapsto D_3[D_4]$ where D_3 and D_4 must be application contexts to ensure that D_2 is an application context. The forking table for (A,T) has only one entry $(A,A,T,\texttt{app}\ [\cdot_1]\ [\cdot_2])$, since an application context D_1 and a top context D_2 can only have different hole pathes, if there is an application where the hole path of D_1 goes through the first argument, while the hole path of D_2 goes through the second argument, the expression above this application must belong to application contexts (to ensure that D_1 is an application context) the context inside the first argument of the application must be an application context (again to ensure that D_1 is an application context), and the context inside the second argument must be a top context (to ensure that D_2 is a top context). For (T,T) there are more entries, since the forking of two top-contexts may happen in an application or in a `letrec`-expression: there are two cases for the application depending on whether the hole path of the first context goes through the first or the second argument, and there are three cases for `letrec`: the hole path of the first context may go through the `in`-expression while the other goes through the `letrec`-environment, or vice versa, or both hole paths go through the environment, but through different bindings. In any case the context above the two parallel holes is a top-context and the contexts below must both be top-contexts.

The semantics of meta-variables is straight-forward except for chain-variables: $Ch[x,s]$ with $cl(Ch) = \mathscr{K}$ stands for $x.\mathtt{d}[s]$ or chains $x.\mathtt{d}_1[(\texttt{var}\ \mathtt{x}_1)];\mathtt{x}_1.\mathtt{d}_2[(\texttt{var}\ \mathtt{x}_2)];\ldots;\mathtt{x}_n.\mathtt{d}_n[s]$ with fresh $\mathtt{x}_i$ and contexts $\mathtt{d},\mathtt{d}_i$ of class $\mathscr{K}$. For expression e, $MV(e)$ denotes the meta-variables of e, $FV(e)$ denotes the free variables, $BV(e)$ denotes the bound variables, and $Var(e) := FV(e) \cup BV(e)$. For a ground context $\mathtt{d}$, $CV(\mathtt{d})$ (the *captured variables*) is the set of variables $\mathtt{x}$ which become bound if plugged into the hole of $\mathtt{d}$. For environment *env*, $LV(env)$ are the let-bound variables in *env*. Let $\sim_{let}$ be the reflexive-transitive closure of permuting bindings in a `letrec`-environment, and $\sim_\alpha$ be the reflexive-transitive closure of combining $\sim_{let}$ and α-equivalence. An LRSX-expression s satisfies the *let variable convention (LVC)* iff a let-bound variable does not occur twice as a binder in the same `letrec`-environment; and s satisfies the *distinct variable convention (DVC)* iff $BV(s)$ and $FV(s)$ are disjoint and all binders bind different variables.

3.2 Rewrite Rules

The left and the right hand side of a standard reduction rule or a program transformation can be represented by LRSX-expressions. However, the rules and the transformations come with additional constraints, for instance, for the garbage collection rules, we need to express that a (part of a) letrec-environment is indeed unused and garbage. We thus constrain expressions by so-called constraint tuples:

Definition 3.3. A *constrained expression* (s,Δ) consists of an LRSX-expression s and a *constraint tuple* $\Delta = (\Delta_1,\Delta_2,\Delta_3)$ such that Δ_1 is a finite set of context variables, called *non-empty context constraints*; Δ_2 is a finite set of environment variables, called *non-empty environment constraints*; and Δ_3 is a finite set of pairs (t,d) where t is an LRSX-expression and d is an LRSX-context, called *non-capture constraints* (NCCs). A ground substitution ρ *satisfies* Δ iff $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$; $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$; and $Var(\rho(t)) \cap CV(\rho(d)) = \emptyset$ for all $(t,d) \in \Delta_3$. The *concretizations* of (s,Δ) are $\gamma(s,\Delta) := \{\rho(s) \mid \rho \text{ is a ground substitution, } \rho(s) \text{ fulfills the LVC, } \rho \text{ satisfies } \Delta\}$[3].

Example 3.4. The constrained expression $(\lambda X.S,(\emptyset,\emptyset,\{(S,\lambda X.[\cdot])\}))$ represents all abstractions that do not use their argument, since the NCC $(S,\lambda X.[\cdot])$ ensures that (w.r.t. instances) the variable X does not occur free in S. The constrained expression $(\texttt{letrec}\ E\ \texttt{in}\ S,(\emptyset,\{E\},\{(S,\texttt{letrec}\ E\ \texttt{in}\ [\cdot])\}))$ represents

[3] In the LRSX Tool constrained expressions are written as "*e* `where` *Constraints*" such that *Constraints* is a list of constraints, where non-empty context constraints are written as `D /= [.]`, non-empty environment constraints are written as `E /= {}`, and non-capture constraints can occur as (s,d), but also as $[env,d]$ representing the NCC $(\texttt{letrec}\ env\ \texttt{in}\ c,d)$ for some constant c.

all letrec-expressions with a non-empty environment which is garbage: the NCC forbids references from S into the environment E. The constrained expression $(C[\texttt{var}\,X],(\{C\},\emptyset,\{(\texttt{var}\,X,C)\}))$ represents ground expressions of the form $d[\texttt{var x}]$ where d is a non-empty context and the occurrence of x in the context hole of d is guaranteed to be a free occurrence.

We have introduced the formalisms that are required to define our representation of standard reduction rules and program transformations. We now introduce the notion of letrec rewrite rules which are rewrite rules on LRSX-expressions, constrained by a constraint tuple, and which have restrictions on the occurrences of meta-variables. The restrictions make the corresponding unification and matching problems easier to solve. Usually, the rules of a program calculus fulfill these restrictions.

Definition 3.5. For $\ell, r \in$ **Exp**, a constraint tuple Δ, $\kappa \in \{SR, \mathsf{T}\}$, a name n, we say that $\ell \xrightarrow{\kappa,n}_\Delta r$ is a *letrec rewrite rule*, if the following conditions hold: (i) $MV(\Delta) \subseteq MV(\ell) \cup MV(r)$; (ii) in each of the expressions ℓ and r, every variable of type S occurs at most twice, and every variable of kind E, Ch, D occurs at most once; and Ch-variables occurring in ℓ must occur in one letrec-environment only; (iii) for any ground substitution ρ that satisfies Δ, $\rho(\ell)$ fulfills the LVC iff $\rho(r)$ fulfills the LVC. A letrec rewrite rule represents the set of ground rewrite rules

$$\gamma(\ell \xrightarrow{\kappa,n}_\Delta r) := \{\rho(\ell) \to \rho(r) \mid \rho \text{ is ground for } \ell, r, \text{ the LVC holds for } \rho(\ell), \rho(r), \rho \text{ satisfies } \Delta\}.$$

For a set $\{\ell \xrightarrow{\kappa,n_i}_\Delta r \mid i = 1,\ldots,m\}$ of letrec rewrite rules, we write $\mathsf{s} \xrightarrow{\kappa,n_i} \mathsf{t}$ if $(\mathsf{s} \to \mathsf{t}) \in \gamma(\ell \xrightarrow{\kappa,n_i}_\Delta r)$ and $\mathsf{s} \xrightarrow{\kappa} \mathsf{t}$ if $\mathsf{s} \xrightarrow{\kappa,n_i} \mathsf{t}$ for some $1 \le i \le m$. We write $\mathsf{s} \;{}_\alpha\!\xrightarrow{\kappa,n_i}\; \mathsf{s}'$ if there exists s'' such that $\mathsf{s} \sim_\alpha \mathsf{s}'' \xrightarrow{\kappa,n_i} \mathsf{s}'$.

Standard reductions are letrec rewrite rules that are always applicable to expressions which fulfill the DVC. Answers are constrained expressions which represent successful programs:

Definition 3.6. A *standard reduction* is a letrec rewrite rule $\ell \xrightarrow{\kappa,n}_\Delta r$ such that the following condition holds: if for ground expressions $\mathsf{s}_1, \mathsf{s}_2$ with $\mathsf{s}_1 \xrightarrow{SR,n} \mathsf{s}_2 \in \gamma(\ell \xrightarrow{\kappa,n}_\Delta r)$, then for all ground expressions t_1, such that $\mathsf{s}_1 \sim_\alpha \mathsf{t}_1$ and t_1 fulfills the DVC, there exists $\mathsf{t}_2 \sim_\alpha \mathsf{s}_2$, such that $\mathsf{t}_1 \xrightarrow{SR,n} \mathsf{t}_2 \in \gamma(\ell \xrightarrow{\kappa,n}_\Delta r)$. An *answer set* Ans is a finite set of constrained expressions (t,Δ) such that if $\mathsf{s} \in \gamma(t,\Delta)$, then for all $\mathsf{s}' \sim_\alpha \mathsf{s}$ such that s' fulfills the DVC we have $\mathsf{s}' \in \gamma(t,\Delta)$. If $\mathsf{s} \in \gamma(t,\Delta)$ for some $(t,\Delta) \in$ Ans and $\mathsf{s}' \sim_\alpha \mathsf{s}$, then s' is called an *answer*. A *program calculus* is a pair (SR, Ans) of a finite set of standard reductions SR and an answer set Ans, such that whenever $\mathsf{s} \xrightarrow{SR,n} \mathsf{s}'$ and s is an answer, then also s' is answer.

Example 3.7. The calculus *Simp* is a program calculus by instantiation the set SR with the standard reductions $(sr,bot),(sr,top),(sr,neg1),(sr,neg2)$ and the answer set Ans by $\{(\top,(\emptyset,\emptyset,\emptyset))\}$. Also the calculus L_{need} is a program calculus where $\mathsf{Ans} := \{(\lambda X.S,(\emptyset,\emptyset,\emptyset)),(\texttt{letrec}\ E\ \texttt{in}\ S,(\emptyset,\{E\},\emptyset))\}$ and SR are all standard reduction rules (partly shown in Fig. 7).

In the LRSX Tool, standard reduction $\ell \xrightarrow{SR,n}_\Delta r$ is written "{SR,n,k} ℓ ==> r where *Constraints*" such that k is a number (the variant of the rule[4]) and *Constraints* are the constraints in Δ written as in constrained expressions. Answers are defined in the LRSX Tool by "ANSWER e where *Constraints*."

For the calculus L_{need}, the conditions on standard reductions hold. An excerpt of the description of L_{need} as input of the LRSX Tool is in Fig. 7, where rules (sr,lbeta) and (sr,lapp) are expressed by three rules each, since the reduction contexts R are unfolded into three cases: the reduction context is an A-context, the reduction context has the hole in the in-part of the letrec, or the hole is in the letrec-environment. Chain-variables are written as Ch^K where K is the context class of the chain. Side conditions of the rules (see Fig. 5) are expressed by constraints. The last two lines define the answers in L_{need}, which are the weak head normal forms, i.e. abstractions perhaps with an outer letrec.

[4] In short representation of rule names, the LRSX Tool unions all variants of a rule of the same name.

```
{SR,lbeta,1}   A[app (\X.S1) S2] ==> A[letrec X=S2 in S1] where (S2,\X.[.])
{SR,lbeta,2}   letrec E in A[app (\X.S1) S2]
               ==> letrec E in A[letrec X=S2 in S1] where E /= {}, (S2,\X.[.])
{SR,lbeta,3}   letrec E; Ch^A[X1,app (\X.S1) S2] in A1[var X1]
               ==> letrec E; Ch^A[X1,letrec X=S2 in S1] in A1[var X1] where (S2,\X.[.])
{SR,lapp,1}    A[app (letrec E in S1) S2] ==> A[letrec E in (app S1 S2)]
                where  E /={},(S2,letrec E in [.])
{SR,lapp,2}    letrec E1 in A[app (letrec E in S1) S2]
               ==> letrec E1 in A[letrec E in (app S1 S2)]
                where E1 /= {},E /={},(S2,letrec E in [.])
{SR,lapp,3}    letrec E1;Ch^A[X,app (letrec E in S1) S2] in A1[var X]
               ==> letrec E1;Ch^A[X,letrec E in app S1 S2] in A1[var X]
                where E/={},(S2,letrec E in [.])
...
ANSWER \X.S
ANSWER letrec E in \X.S where E /= {}
```

Figure 7: Some standard reductions and answers for L_{need} as input for the LRSX Tool

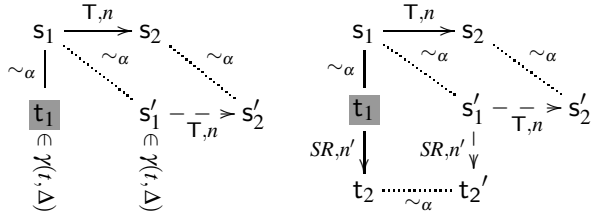


Figure 8: Illustration of Cond. 1 and 2 in Def. 3.9: solid lines are given relations, dotted / dashed lines are existentially quantified relations, t_1 fulfills the DVC.

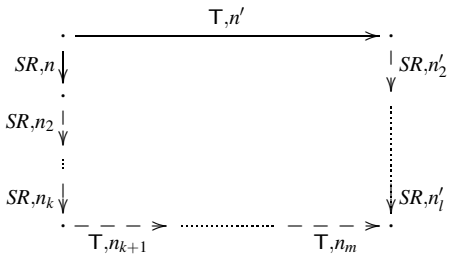


Figure 9: Representation of a forking diagram

Definition 3.8. For a program calculus $(\mathsf{SR},\mathsf{Ans})$, a ground expression s_0 *converges* (written $s_0{\downarrow}$) iff there exists a sequence $s_0 \xrightarrow[\alpha]{SR} s_1 \xrightarrow[\alpha]{SR} \cdots \xrightarrow[\alpha]{SR} s_k$ where s_k is an answer and $k \geq 0$. We write $s \leq_{\downarrow} t$ iff $s{\downarrow} \implies t{\downarrow}$ ($\leq_{\downarrow}$ is called *convergence approximation*), and $s \sim_{\downarrow} t$ iff $s \leq_{\downarrow} t$ and $t \leq_{\downarrow} s$ ($\sim_{\downarrow}$ is called *convergence equivalence*). If for all contexts C we have $C[s] \leq_{\downarrow} C[t]$, then we write $s \leq_c t$ and say that t contextually approximates s. Expressions s,t are contextually equivalent ($s \sim_c t$) if $s \leq_c t$ and $t \leq_c s$.

Meta transformations are letrec rewrite rules that fulfill some form of stability w.r.t. α-renaming. These conditions on meta transformations allow us to inspect overlaps between transformations and standard reductions or answers *without* considering α-renaming steps. I.e., they guarantee that inspecting overlaps of the form $s_1 \xleftarrow{SR} s_2 \xrightarrow{T} s_3$ is sufficient, and hence inspecting overlaps of the form $s_1 \xleftarrow{SR}_{\alpha} s_2 \xrightarrow[\alpha]{T} s_3$, where the α-renaming part of $s_1 \xleftarrow{SR}_{\alpha} s_2$ is non-trivial, is not necessary (see Appendix A for a soundness proof of the diagram technique which also formalizes this aspect).

Definition 3.9. A letrec rewrite rule with $\kappa = \mathsf{T}$ is a *meta transformation*, if the following conditions hold (see also Fig. 8): for all s_1,s_2,t_1 with $s_1 \xrightarrow{\mathsf{T},n} s_2$, $s_1 \sim_\alpha t_1$, such that t_1 fulfills the DVC: 1. If $t_1 \in \gamma(t,\Delta)$ for some $(t,\Delta) \in \mathsf{Ans}$, then there exists $s'_1 \in \gamma(t,\Delta)$ such that $s'_1 \sim_\alpha s_1$ and $s'_1 \xrightarrow{\mathsf{T},n} s'_2$ with $s'_2 \sim_\alpha s_2$. 2. If $t_1 \xrightarrow{SR,n'} t_2$, then there exist $s'_1 \sim_\alpha s_1$, $s'_2 \sim_\alpha s_2$, $t'_2 \sim_\alpha t_2$ such that $s'_1 \xrightarrow{\mathsf{T},n} s'_2$, and $s'_1 \xrightarrow{SR,n'} t'_2$.

A meta transformation $\ell \xrightarrow{\mathsf{T},n}_{\Delta} r$ is *correct* iff $\gamma(\ell \xrightarrow{\mathsf{T},n}_{\Delta} r) \subseteq \sim_c$. A meta transformation $\ell \xrightarrow{\mathsf{T},n}_{\Delta} r$ is called *overlapable* if no *Ch*-variable occurs in ℓ and r and the transformation is closed w.r.t. a sufficient

context class for $\sim_c$, i.e. $\mathsf{s} \xrightarrow{\mathsf{T},n} \mathsf{t}$, $\mathsf{s} \leq_{\downarrow} \mathsf{t}$ imply $\mathsf{s} \leq_c \mathsf{t}$.

A sufficient criterion for Conditions (1) and (2) from Definition 3.9 is that applicability of a transformation to an expression s implies applicability of the transformation to all α-renamed expressions $s' \sim_\alpha s$ that fulfill the DVC:

Proposition 3.10. *Let* $(\mathsf{SR},\mathsf{Ans})$ *be a program calculus and* $s \xrightarrow{\mathsf{T},n}_\Delta t$ *be a letrec rewrite rule such that no Ch-variable occurs in* ℓ *and* r *and the transformation is closed w.r.t. a sufficient context class for contextual equivalence. Assume that* $\mathsf{s}_1 \xrightarrow{\mathsf{T},n} \mathsf{s}_2$ *implies that for all* $\mathsf{s}'_1 \sim_\alpha \mathsf{s}_1$ *such that* s'_1 *fulfills the DVC also* $\mathsf{s}'_1 \xrightarrow{\mathsf{T},n} \mathsf{s}'_2$ *for some* $\mathsf{s}'_2 \sim_\alpha \mathsf{s}_2$. *Assume also that* $\mathsf{s}_1 \xrightarrow{\mathsf{T},n} \mathsf{s}_2$ *for* $\mathsf{s}_1 \in \gamma(\mathsf{Ans})$ *implies that for all* $\mathsf{s}'_1 \sim_\alpha \mathsf{s}_1$ *also* $\mathsf{s}'_1 \xrightarrow{\mathsf{T},n} \mathsf{s}'_2$ *holds for some* $\mathsf{s}'_2 \sim_\alpha \mathsf{s}_2$. *Then* $s \xrightarrow{\mathsf{T},n}_\Delta t$ *is overlapable.*

In L_{need}, the criterion holds for most of the considered transformations. An exception is the reversed copy transformation, (e.g. the reversal of $\xrightarrow{\text{cp-in}}$ in Fig. 5). It violates the criterion in Proposition 3.10, since all ground instances of the left hand side violate the DVC. However, Conditions (1) and (2) from Definition 3.9 hold, since two occurrences of $\lambda v.e$ do not forbid the application of a standard reduction.

Meta transformations $\ell \xrightarrow{\mathsf{T},n}_\Delta r$ are written in the LRSX Tool as "`{`n`,`k`}` ℓ `==>` r `where` *Constraints*" where k is a non-negative integer representing the variant of the rule. For the calculus L_{need} a context lemma [17] holds, which shows that top contexts are a sufficient class for $\sim_c$, thus it suffices to consider the closure of garbage collection w.r.t. top contexts. We can represent the rules for garbage collection as:

```
{gcT,1} T[letrec E1;E2 in S] ==> T[letrec E1 in S]
         where E1 /= {}, E2 /= {}, [E1,letrec E2 in [.]], (S,letrec E2 in [.])
{gcT,2} T[letrec E in S] ==> T[S]  where E /= {}, (S,letrec E in [.])
```

4 Computing Diagrams and Automated Induction

For proving $\gamma(gcT) \subseteq \leq_{\downarrow}$, we have to compute all overlaps between the left hand side of (gcT) and an answer (called *answer overlaps*[5]), and between the left hand sides of (gcT) and a standard reduction (called *forking overlaps*)[6]. Clearly, computing the overlaps cannot be done using the concretizations w.r.t. γ, but has to be done on the meta-syntax, i.e. by unifying the left hand sides of the meta-transformation with the left hand sides of the standard reductions and the answers, respecting the constraint tuples corresponding to the rules. An appropriate unification algorithm for `LRSX` was developed in [18] and implemented in the LRSX Tool. Calling the tool produces 99 (93, resp.) overlaps of (gcT,1) ((gcT,2) resp.) with all standard reductions and answers. For joining the overlaps we have to apply standard reductions and transformation rules to the constrained expressions (again on the meta-syntax) of the overlaps until a common successor is found. For an answer s and an answer overlap $s \xrightarrow{\mathsf{T},n'} t$, a *join* is a sequence $t_k \xleftarrow{SR,n_k}_\alpha \cdots \xleftarrow{SR,n_1}_\alpha t$ where $k \geq 0$ and $t_k \in \gamma(\mathsf{Ans})$. For a forking overlap $s_1 \xleftarrow{SR,n} t \xrightarrow{\mathsf{T},n'} t_1$, a *join* is a sequence

$$s_1 \xrightarrow[\alpha]{SR,n_2} \cdots \xrightarrow[\alpha]{SR,n_k} s_k \xrightarrow[\alpha]{\mathsf{T},n_{k+1}} \cdots \xrightarrow[\alpha]{\mathsf{T},n_m} s_m \sim_\alpha t_l \xleftarrow{SR,n'_l}_\alpha \cdots \xleftarrow{SR,n'_2}_\alpha t_1$$

where $m,k,l \geq 1$ and $k > 1$ is only allowed if $(\mathsf{SR},\mathsf{Ans})$ is deterministic[7]. The forking overlap together with a join builds a *forking diagram* which can be depicted as shown in Fig. 9 (where steps from the

[5] Internally, answer overlaps are computed as overlaps with rules $\ell \xrightarrow{answer} ans$ for $\ell \in \mathsf{Ans}$ and a new constant ans.

[6] In the LRSX Tool the commands to overlap the left hand sides with all standard reductions are `overlap (gcT,1).l all` and `overlap (gcT,2).l all`.

[7] For each ground expression s, there exists at most one t such that $\mathsf{s} \xrightarrow{SR} \mathsf{t} \in \gamma(\mathsf{SR})$.

overlap are written with solid arrows, and (existentially quantified) steps of the join are written with dashed arrows). Similarly, for an answer overlap together with its join is called an *answer diagram*.

Applying letrec rewrite rules uses a matching algorithm for `LRSX` (see [13]). A peculiarity of the matching problem is, that constrained expressions of the overlap have to be matched against meta-expressions from the rewrite rule which also come with constraint tuples. Thus the algorithm has to guarantee that the given constraints imply the needed constraints before returning a matcher. Additionally, the rewrite mechanism has to guarantee completeness w.r.t. ground instances, i.e. each rewrite step on the meta-level (applying meta rewrite rules to constrained expressions) must also be possible for all ground instances. Our tool uses an iterative and depth-bounded depth-first search to bound the number of applied transformations and reductions. Since sometimes no join is found, since a possible rewriting requires more knowledge on the (non-)emptiness of environment and context variables, the LRSX Tool uses backtracking: if no join is found for an overlap, then first a case distinction for context variables in the problem is done (whether they are empty or non-empty) and then the case distinction is done for environment variables. As a further feature, in the LRSX Tool the search space for joins can be limited: using the `ignore`-primitive of the tool one can forbid to use some transformations at all for the search for joins, and with the `restrict`-primitive the number of allowed uses of a transformation can be bounded.

For checking if a join is found, we have to test equivalence of constrained expressions. A simple check is testing $\sim_{let}$, but however, also the constraint tuples have to be checked. We omit the more complicated check, but in [14] a sound and complete check for proving equivalence of constrained expressions can be found. A key technique in the check is to split non-capture constraints (s,d) into *atomic* non-capture constraints which are pairs (u,v) such that u,v are variables or meta-variables. The split is done by collecting the variables and meta-variables appearing in s and in d. A ground substitution ρ *satisfies* an atomic NCC (u,v) iff $Var(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$ where $CV_A(\mathtt{x}) = \{\mathtt{x}\}$ for all variables $\mathtt{x}$ and $CV_A(r) = CV(r)$ for all other constructs r. Since ρ satisfies (s,d) iff it satisfies all split NCCs, the computations for checking equivalence of constraints can be done on the sets of atomic NCCs.

The `join`-command of the LRSX Tool tries to join the found overlaps and to compute forking and answer diagrams: The diagrams are rewrite rules where the left hand side represents the overlap and the right hand represents the join, where on both sides the diagrams are abstracted from the concrete expressions (and thus they represent string rewrite systems where the alphabet are names or reductions and transformations and the abstract symbol `<-ANSWER-`). For our example, the computed forking diagrams and answer diagrams (in textual representation, and condensed form) are shown in Fig. 11 and a pictorial representation of the forking diagrams is in Fig. 10. Here unions of rules are used (which are also supported in the LRSX Tool): `(SR,lbeta)` is the union of `(SR,lbeta,1)`, `(SR,lbeta,2)`, and `(SR,lbeta,3)`. `(SR,lapp)` is the union of `(SR,lapp,1)`, `(SR,lapp,2)`, and `(SR,lapp,3)`, `(SR,cp)` is the union of the rules representing (sr,cp-in) and (sr,cp-e), `(SR,llet)` is the union of the rules representing (sr,llet-in) and (sr,llet-e) (see Fig. 5), and `(SR,lll)` is the union of `(SR,llet)` and `(SR,lapp)`. In a pen-and-paper proof of $\gamma(gcT) \subseteq \leq_{\downarrow}$, an induction on the length of a converging reduction sequence $s \xrightarrow{SR,*} s'$ for s with $s \xrightarrow{gcT} t$ is used to show that t converges. The induction base is covered by the answer diagrams, and for the induction step, let $s \xrightarrow{SR} s_1 \xrightarrow{SR,*} s'$. Applying a forking diagram to $s_1 \xleftarrow{SR} s \xrightarrow{gcT} t$ shows existence of some t' with $s_1 \xrightarrow{gcT} t' \xleftarrow{SR} t$ or $s_1 \xrightarrow{gcT} t' = t$ and by the induction hypothesis $t'\downarrow$ which also implies $t\downarrow$. This induction (even with more complex induction measures) can be automatized by interpreting the answer and forking diagrams as term rewrite system and by showing (innermost) termination of them (see [11]). From the obtained answer and forking diagrams for (gcT), the LRSX Tool generates the term rewrite system shown in Fig. 12 which can be proved to be innermost terminating using the prover AProVE and the certifier CeTA.

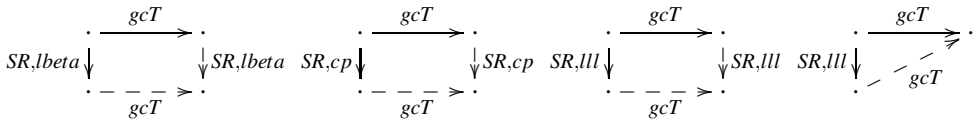


Figure 10: Diagrams for (gcT), pictorial

```
<-SR,lbeta- . -gcT-> ~~> -gcT-> . <-SR,lbeta-
   <-SR,cp- . -gcT-> ~~> -gcT-> . <-SR,cp-
  <-SR,lll- . -gcT-> ~~> -gcT-> . <-SR,lll-
  <-SR,lll- . -gcT-> ~~> -gcT->
  <-ANSWER- . -gcT-> ~~> <-ANSWER-
```

Figure 11: Diagrams for (gcT), textual

```
gcT(SRlbeta(x)) -> SRlbeta(gcT(x))
gcT(SRcp(x)) -> SRcp(gcT(x))
gcT(SRlll(x)) -> SRlll(gcT(x))
gcT(SRlll(x)) -> gcT(x)
gcT(Answer) -> Answer
```

Figure 12: Obtained TRS for (gcT)

5 Extended Techniques and Limitations of the Method

Our example to prove $\gamma(gcT) \subseteq \leq_{\downarrow}$ is quite simple. Unification and matching for LRSX-expressions and usual term rewrite systems for the automated induction are successful. However, the LRSX Tool provides more sophisticated techniques that are for instance required when proving the remaining part, i.e. $\gamma(gcT) \subseteq \geq_{\downarrow}$, to complete the correctness proof of garbage collection. First observe that the diagram technique works as before with the difference that the reversal of (gcT) is used (i.e. with writing $(gcT)^-$ for reversing the transformation (gcT) we have to show $\gamma((gcT)^-) \subseteq \leq_{\downarrow}$). However, this means that we have to overlap left hand sides of standard reductions and answers with *right* hand sides of (gcT). The obtained overlaps are called answer and *commuting diagrams*. Computing the overlaps results in 99 overlaps for (gcT,1) and 203 overlaps for (gcT,2). However, using the presented techniques for computing joins fails. An overlap (we omit the constraints) which cannot be joined is

$$\begin{array}{ccc} A[(\lambda X.S)\ T[\texttt{letrec}\ E_1\ \texttt{in}\ S']] & \xleftarrow{gcT,1} & A[(\lambda X.S)\ T[\texttt{letrec}\ E_1; E_2\ \texttt{in}\ S']] \\ \downarrow{\scriptstyle SR,lbeta,1} & & \\ A[\texttt{letrec}\ X.T[\texttt{letrec}\ E_1\ \texttt{in}\ S']\ \texttt{in}\ S] & & \end{array}$$

The automated method cannot apply a (SR,lbeta)-reduction to the upper-right expression, since it cannot infer that variable X does not occur in E_2. However, this problem can be solved by α-renaming the expression such that the DVC holds. That is why symbolic α-renaming (see [12]) is built into the LRSX Tool which is quite more complex than usual α-renaming, since it has to be performed on the meta syntax, e.g. internally symbolic renamings of the form $\alpha \cdot S$ are required. Even with α-renaming, the LRSX Tool cannot join all overlaps. E.g., for the overlap (we omit the constraints) $A[\texttt{letrec}\ X.S'\ \texttt{in}\ S] \xleftarrow{SR,lbeta,1} A[(\lambda X.S)\ S'] \xleftarrow{gcT,2} A[(\texttt{letrec}\ E\ \texttt{in}\ (\lambda X.S))\ S']$ a meta-argument is required to close the overlap stating that the standard reduction moves the environment E to the top of the expression, i.e. a sequence $A[(\texttt{letrec}\ E\ \texttt{in}\ (\lambda X.S))\ S'] \xrightarrow{SR,lll,+} \texttt{letrec}\ E\ \texttt{in}\ A[(\lambda X.S))\ S']$ where $\xrightarrow{SR,lll,+}$ is the transitive closure of $\xrightarrow{SR,lll}$. In the LRSX Tool such transitive closures can be defined and with these rules it is able to compute a complete set of commuting diagrams for the (gcT)-transformation. A pictorial representation of the commuting diagrams for $a \in \{lbeta, cp, lll\}$ is shown in Fig. 13.

The automated induction has to treat the transitive closure in the rules. A naive encoding leads to term rewrite systems with infinitely many rules. The LRSX Tool generates a term rewrite system with free variables on the right hand sides (or alternatively integer term rewrite systems, see [11, 4]) where

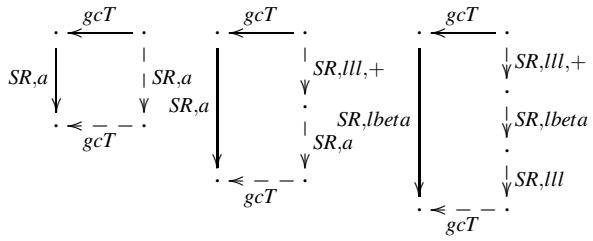


Figure 13: Commuting Diagrams for (gcT)

```
gcT(SRlbeta(x)) -> W24(k,x)
W24(s(k),x) -> SRlll(W24(k,x))
W24(s(k),x) -> SRlll(SRlbeta(gcT(x)))
```

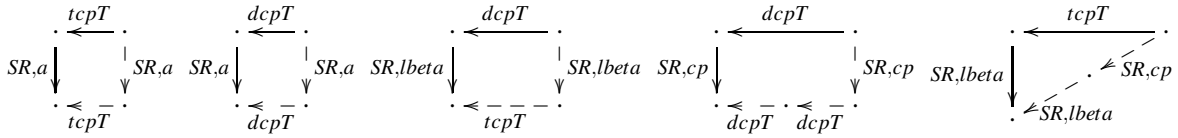
Figure 14: Term rewrite rules for the 2nd diagram

these variables are interpreted as variables representing constructors. Every transitive closure is encoded as a guessing of the number of steps it represents. E.g., the second diagram is encoded by three term rewrite rules in Fig. 14. The termination prover AProVE and the certifier CeTA support such termination problems with free variables on right-hand sides interpreted as arbitrary constructor term. For (gcT), innermost termination can be proved and certified.

Now consider the transformations (cp-in) and (cp-e) from Fig. 5 closed by top-contexts. Computing commuting diagrams and deriving the corresponding term rewrite system results in the system

```
cpT(SRlbeta(x)) -> SRlbeta(cpT(x))        cpT(SRcp(x)) -> SRcp(cpT(cpT(x)))
  cpT(SRlll(x)) -> SRlll(cpT(x))       cpT(SRlbeta(x)) -> SRcp(SRlbeta(x))
   cpT(SRcp(x)) -> SRcp(cpT(x))
```

which is non-terminating. If we split (cpT) into transformations where the copy target is a top-context (tcpT) and where the target is below an abstraction (dcpT), then the diagram set becomes



and termination of the corresponding TRS can be proved.

We conclude this section by explaining situations for program calculi and program transformations that cannot be handled by the current version of the LRSX Tool. The underlying meta language has no support for substitutions, i.e. usual β-reduction $(\lambda x.s)\ t \to s[t/x]$ can only be represented by encoding explicit substitutions (for instance, by using the `letrec`-construct). Languages which use an equational theory to equate programs (for instance, structural congruence in the π-calculus [8] or in the CHF-calculus[16]) are not supported at the moment, since this would require unification and matching to handle the equational theory. Also program transformations with more complicated side-conditions (for instance, those using strictness information) can not be represented in the tool, since only rules that can be constrained the constraint tuples can be represented. Finally, the occurrence restrictions on meta-variables and the conditions on program transformations clearly forbid some program transformations. For instance, we do not allow program transformations that use chain-variables, for calculi which use chain-variables in the standard reduction rules.

6 Implementation and Experiments

The Haskell-implementation of the automated diagram method to prove correctness of program transformation is available as a Cabal-package from http://goethe.link/LRSXTOOL61. We tested our implementation with three different program calculi and a lot of program transformations. The tested calculi

	# overlaps		# meta joins		# meta joins with α-renaming		diagram computation time
	forking	answer	forking	answer	forking	answer	
Calculus L_{need} (11 SR rules, 16 transformations, 2 answers)							
$\rightarrow$	2215	27	5398	27	93	0	48 secs.
$\leftarrow$	2963	38	7235	38	1399	3	116 secs.
Calculus L_{need}^{+seq} (17 SR rules, 18 transformations, 2 answers)							
$\rightarrow$	4869	29	14700	29	143	0	149 secs.
$\leftarrow$	6394	43	18046	43	2374	3	255 secs.
Calculus LR (76 SR rules, 43 transformations, 17 answers)							
$\rightarrow$	85455	1586	389678	1586	73601	0	$\sim$ 19 hours
$\leftarrow$	105053	2280	426664	2440	93075	155	$\sim$ 16 hours

Table 1: Statistics of executing the LRSX Tool

are the calculus L_{need} [19] – a minimal call-by-need lambda calculus with `letrec` – the calculus L_{need}^{+seq} which extends L_{need} by the `seq`-operator, where `seq` e_1 e_2 first evaluates the first argument e_1 and after obtaining a successful result it evaluates argument e_2, and the calculus LR [20] which extends L_{need}^{+seq} by data constructors for lists, booleans and pairs together with corresponding case-expressions, and can be seen as an untyped core language of Haskell. The tested program transformations include all calculus reductions which can be summarized as "partial evaluation", several copying transformations and rules for removing garbage and inlining of let-bindings which are referenced only once.

Our experimental results are in Table 1, where we also list the numbers of standard reductions, transformations, and answers in the input. The table shows the numbers of computed overlaps, corresponding joins (which is higher due to the branching in unsuccessful cases), joins which use the α-renaming procedure. The row marked with $\rightarrow$ represent the forking diagrams, and $\leftarrow$ represent the reversed transformations, i.e. commuting diagrams. In all cases, termination of the termination problems was proved by AProVE and certified by CeTA. The last column lists the execution time[8] for calculating the overlaps and the joins. With increasing numbers of rules, transformations, and syntactic constructs the computation time increases, due to the combinatorial explosion. The time to compute joins for commuting diagrams in LR is higher than for computing forking diagrams, since we put more effort in optimizing the commuting diagram computation (by avoiding unusual search paths).

7 Conclusion

We presented a system (the LRSX Tool) to automatically prove correctness of program transformations. We illustrated its use by an example and discussed peculiarities of its design and its implementation. By providing the results of experiments, we demonstrated the success of the method and the tool.

Acknowledgments. We thank René Thiemann for support on AProVE and CeTA. We also thank the anonymous reviewers of WPTE 2018 for their valuable comments.

[8] Tests ran on a system with Intel i7-4790 CPU 3.60GHz, 8 GB memory using GHC's `-N` option for parallel execution

References

[1] Z. M. Ariola & M. Felleisen (1997): *The Call-By-Need lambda Calculus*. *JFP* 7(3), pp. 265–301.

[2] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky & P. Wadler (1995): *A call-by-need lambda calculus*. In: *POPL 1995*, ACM, pp. 233–246, doi:10.1145/199448.199507.

[3] F. Baader & T. Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1017/CBO9781139172752.

[4] C. Fuhs, J. Giesl, M.Plücker, P. Schneider-Kamp & S. Falke (2009): *Proving Termination of Integer Term Rewriting*. In: *RTA 2009*, *LNCS* 5595, Springer, pp. 32–47, doi:10.1007/978-3-642-02348-4_3.

[5] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski & R. Thiemann (2014): *Proving Termination of Programs Automatically with AProVE*. In: *IJCAR 2014*, *LNCS* 8562, Springer, pp. 184–191, doi:10.1007/978-3-319-08587-6_13.

[6] A. Jez (2014): *Context Unification is in PSPACE*. In: *ICALP 2014, Part II*, *LNCS* 8573, Springer, pp. 244–255, doi:10.1007/978-3-662-43951-7_21.

[7] E. Machkasova & F. A. Turbak (2000): *A Calculus for Link-Time Compilation*. In: *ESOP 2000*, *LNCS* 1782, Springer, pp. 260–274, doi:10.1007/3-540-46425-5_17.

[8] R. Milner (1999): *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[9] J. H. Morris (1968): *Lambda-Calculus Models of Programming Languages*. Ph.D. thesis, MIT.

[10] G. D. Plotkin (1975): *Call-by-name, call-by-value, and the lambda-calculus*. *Theoret. Comput. Sci.* 1, pp. 125–159, doi:10.1016/0304-3975(75)90017-1.

[11] C. Rau, D. Sabel & M. Schmidt-Schauß (2012): *Correctness of Program Transformations as a Termination Problem*. In: *IJCAR 2012*, *LNCS* 7364, Springer, pp. 462–476, doi:10.1007/978-3-642-31365-3_36.

[12] D. Sabel (2017): *Alpha-renaming of Higher-order Meta-expressions*. In: *PPDP 2017*, ACM, pp. 151–162, doi:10.1145/3131851.3131866.

[13] D. Sabel (2017): *Matching of Meta-Expressions with Recursive Bindings*. In: *Informal Proceedings of UNIF 2017*. Available at unif-workshop.github.io/UNIF2017/papers/UNIF_2017_paper_2.pdf.

[14] D. Sabel (2017): *Rewriting of Higher-Order Meta-Expressions with Recursive Bindings*. Frankfurter Informatik-Berichte 2017-1, Goethe-University Frankfurt. Available at d-nb.info/1136368175/34.

[15] D. Sabel & M. Schmidt-Schauß (2008): *A Call-by-Need Lambda-Calculus with Locally Bottom-Avoiding Choice: Context Lemma and Correctness of Transformations*. *Math. Structures Comput. Sci.* 18(03), pp. 501–553, doi:10.1017/S0960129508006774.

[16] D. Sabel & M. Schmidt-Schauß (2011): *A contextual semantics for concurrent Haskell with futures*. In: *PPDP 2011*, ACM, pp. 101–112, doi:10.1145/2003476.2003492.

[17] M. Schmidt-Schauß & D. Sabel (2010): *On generic context lemmas for higher-order calculi with sharing*. *Theoret. Comput. Sci.* 411(11-13), pp. 1521 – 1541, doi:10.1016/j.tcs.2009.12.001.

[18] M. Schmidt-Schauß & D. Sabel (2016): *Unification of Program Expressions with Recursive Bindings*. In: *PPDP 2016*, ACM, pp. 160–173, doi:10.1145/2967973.2968603.

[19] M. Schmidt-Schauß, D. Sabel & E. Machkasova (2010): *Simulation in the Call-by-Need Lambda-Calculus with letrec*. In: *RTA 2010*, *LIPIcs* 6, Schloss Dagstuhl, pp. 295–310, doi:10.4230/LIPIcs.RTA.2010.295.

[20] M. Schmidt-Schauß, M. Schütz & D. Sabel (2008): *Safety of Nöcker's Strictness Analysis*. *JFP* 18(04), pp. 503–551, doi:10.1017/S0956796807006624.

[21] R. Thiemann & C. Sternagel (2009): *Certification of Termination Proofs Using CeTA*. In: *TPHOLs 2009*, *LNCS* 5674, Springer, pp. 452–468, doi:10.1007/978-3-642-03359-9_31.

[22] J. B. Wells, D. Plump & F. Kamareddine (2003): *Diagrams for Meaning Preservation*. In: *RTA 2003*, *LNCS* 2706, Springer, pp. 88 –106, doi:10.1007/3-540-44881-0_8.

[23] A. K. Wright & M. Felleisen (1994): *A Syntactic Approach to Type Soundness*. *Inf. Comput.* 115(1), pp. 38–94, doi:10.1006/inco.1994.1093.

A Soundness of the Diagram Method

We show soundness of the diagram method. Since we sometimes use slightly more general formulations of program transformations for computing joins (but not for computing overlaps), we use two sets of meta transformations. Let $(\mathsf{SR},\mathsf{Ans})$ be a program calculus, OTR be a set of overlapable meta transformations, and $\mathsf{TR} \supseteq \mathsf{OTR}$ be a set of meta transformations such that for each $(\ell \xrightarrow{\mathsf{T},n}_\Delta r) \in \mathsf{TR}$ there exists $(\ell \xrightarrow{\mathsf{T},n'}_\Delta r) \in \mathsf{OTR}$ with $\gamma(\ell \xrightarrow{\mathsf{T},n}_\Delta r) \subseteq \gamma(\ell \xrightarrow{\mathsf{T},n'}_\Delta r)$ (we say that n' *subsumes* n *w.r.t.* γ). A set of forking and answer diagrams is *complete* for a set OTR iff for all forking overlaps of transformations in OTR with standard reductions and every answer overlap, an applicable diagram is in the set. Applicabilty means that the concrete overlap is an instance of the overlap described by the diagram and that the existentially quantified expressions, reductions, and transformations can accordingly be instantiated.

A set of forking and answer diagrams can be viewed as a string rewrite system (that replaces the overlap by the join). In [11] it was shown that proving termination of the string rewrite system with infinitely many rules can be automated by using termination provers for term rewrite systems to show termination of the corresponding integer term rewrite system, or term rewrite system with free variables on the right hand side that represent arbitrary constructor terms. We do not repeat this technqiue here, and formulate our soundness result in terms of the string rewrite system which is induced by the diagrams:

Theorem A.1. *If a complete set of forking and answer diagrams for* OTR *is terminating as a string rewrite system, then all* $\ell \xrightarrow{\mathsf{T},n}_\Delta r \in \mathsf{TR}$ *are convergence equivalent.*

Proof. Since transformations in TR are subsumed by the transformations in OTR it is sufficient to consider $\ell \xrightarrow{\mathsf{T},n}_\Delta r \in \mathsf{OTR}$. Assume that $\mathsf{s} \xrightarrow{\mathsf{T},n} \mathsf{t}$ and $\mathsf{s}{\downarrow}$. Then there exists a sequence $\mathsf{s}'_k \sim_\alpha \mathsf{s}_k \xleftarrow{SR}_\alpha \cdots \xleftarrow{SR}_\alpha \mathsf{s} \xrightarrow{\mathsf{T},n} \mathsf{t}$ where $\mathsf{s}'_k \in \gamma(\mathsf{Ans})$. We apply modifications to the sequence and replace overlaps by joins according to the following rules:

1. If the sequence contains a transformation step $\mathsf{s}_1 \xrightarrow{\mathsf{T},n'} \mathsf{s}_2$ where $\xrightarrow{\mathsf{T},n'}_{\Delta'} \in (\mathsf{TR} \setminus \mathsf{OTR})$, then there exists $\xrightarrow{\mathsf{T},n''}_{\Delta''} \in \mathsf{OTR}$ with $\mathsf{s}_1 \xrightarrow{\mathsf{T},n'} \mathsf{s}_2 \in \gamma(\xrightarrow{\mathsf{T},n''}_{\Delta''})$. Replace $\mathsf{s}_1 \xrightarrow{\mathsf{T},n'} \mathsf{s}_2$ by $\mathsf{s}_1 \xrightarrow{\mathsf{T},n''} \mathsf{s}_2$.
2. If the sequence contains a step $\mathsf{s}_1 \xleftarrow{SR,n'}_\alpha \mathsf{s}_2$, i.e. $\mathsf{s}_1 \xleftarrow{SR,n'} \mathsf{s}'_2 \sim_\alpha \mathsf{s}_2$, and s'_2 does not fulfill the DVC, then replace s'_2 by an expression $\mathsf{s}''_2 \sim_\alpha \mathsf{s}'_2$ such that s''_2 fulfills the DVC. By the definition of standard reductions, the standard reduction $\mathsf{s}'_1 \xleftarrow{SR,n'} \mathsf{s}''_2$ with $\mathsf{s}'_1 \sim_\alpha \mathsf{s}_1$ exists. Replace $\mathsf{s}_1 \xleftarrow{SR,n'} \mathsf{s}'_2 \sim_\alpha \mathsf{s}_2$ by $\mathsf{s}_1 \sim_\alpha \mathsf{s}'_1 \xleftarrow{SR,n'} \mathsf{s}''_2 \sim_\alpha \mathsf{s}_2$.
3. If the sequence contains $\mathsf{s}_1 \xleftarrow{SR}_\alpha \mathsf{s}_2 \;{}_\alpha\!\xrightarrow{SR} \mathsf{s}_3$, then the calculus is deterministic and thus $\mathsf{s}_1 \sim_\alpha \mathsf{s}_3$ holds. Replace the $\mathsf{s}_1 \xleftarrow{SR}_\alpha \mathsf{s}_2 \;{}_\alpha\!\xrightarrow{SR} \mathsf{s}_3$ by $\mathsf{s}_1 \sim_\alpha \mathsf{s}_3$.
4. If the sequence has a prefix $\mathsf{s}_1 \;{}_\alpha\!\xrightarrow{SR} \mathsf{s}_3$ where s_1 is an answer, then the calculus is deterministic and s_3 is an answer and we replace the prefix $\mathsf{s}_1 \;{}_\alpha\!\xrightarrow{SR} \mathsf{s}_3$ by s_3.
5. Subsequences $\mathsf{s}_1 \sim_\alpha \mathsf{s}_2 \sim_\alpha \mathsf{s}_3$ are replaced by $\mathsf{s}_1 \sim_\alpha \mathsf{s}_3$.
6. If the left-most expression of the sequence is $\mathsf{s}_1 \in \gamma(\mathsf{Ans})$ and does not fulfill the DVC, then replace s_1 by $\mathsf{s}'_1 \sim_\alpha \mathsf{s}_1$ such that s'_1 fulfills the DVC. Due to our assumption on answers, $\mathsf{s}'_1 \in \gamma(\mathsf{Ans})$.

7. If the sequence has a prefix $t_1 \sim_\alpha s_1 \xrightarrow{\mathsf{T},n'} s_2$, where t_1 fulfills the DVC and $t_1 \in \gamma(\mathsf{Ans})$, then first apply Condition (1) of Definition 3.9, i.e. replace the prefix by $t_1 \sim_\alpha s_1' \xrightarrow{\mathsf{T},n} s_2' \sim_\alpha s_2$ where $s_1' \in \gamma(\mathsf{Ans})$ and $s_1' \sim_\alpha t$. Since the set of answer diagrams is complete, there is an answer diagram that allows us to replace the answer overlap $s_1' \xrightarrow{\mathsf{T},n} s_2'$ by the corresponding join.
8. If the sequence contains $t_2 \xleftarrow{SR,n'} t_1 \sim_\alpha s_1 \xrightarrow{\mathsf{T},n''} s_2$, then t_1 fulfills the DVC (by the modification in item 2) and we can use Condition 2 of Definition 3.9 and replace $t_2 \xleftarrow{SR,n'} t_1 \sim_\alpha s_1 \xrightarrow{\mathsf{T},n''} s_2$ by $t_2 \sim_\alpha t_2' \xleftarrow{SR,n'} s_1' \xrightarrow{\mathsf{T},n''} s_2' \sim_\alpha s_2$. Since the set of forking diagrams is complete, we can apply a diagram in the set and replace the forking overlap $t_2' \xleftarrow{SR,n'} s_1' \xleftarrow{\mathsf{T},n''} s_2'$ by its join.

The modifications show that we can replace overlaps by joins until the sequence is of the form $s_n \xleftarrow{SR}_\alpha \cdots \xleftarrow{SR}_\alpha t$. Termination of the string rewrite system and the observation that ${}_\alpha\xrightarrow{SR}$-reductions which are introduced by joins can always be removed by the modifications (3) and (4), shows that the replacement together with the modifications terminates. Since, the left end of the sequence is always an expression in $\gamma(\mathsf{Ans})$, this shows $t{\downarrow}$. □

B The Simple Example

We provide the input for the LRSX Tool for the calculus *Simple* and the correctness proof of transformation (top). Note that $\bot$ is represented by `bot`, $\top$ by `top`, $\neg$ by `neg`, and $\wedge$ by `cap` (written prefix).

```
-- file: simple.inp
-- Evaluation contexts A and arbitrary contexts C
define A ::= [.] | (cap A S) | (neg A)
define C ::= [.] | (cap C S) | (cap S C) | (neg C)
-- The prefix table and the forking table
declare prefix A A = (A,A)
declare prefix A C = (A,C)
declare prefix C A = (A,A)
declare prefix C C = (C,C)
declare fork   A C = (A,A,C,(cap [.1] [.2]))
declare fork   C C = (C,C,C,(cap [.1] [.2]))
declare fork   C C = (C,C,C,(cap [.2] [.1]))
declare fork   C A = (A,C,A,(cap [.2] [.1]))
-- standard reduction and answers
{SR,bot}   A[cap bot S] ==> A[bot]
{SR,top}   A[cap top S] ==> A[S]
{SR,neg,1} A[neg top]   ==> A[bot]
{SR,neg,2} A[neg bot]   ==> A[top]
ANSWER top
-- our example transformation:
{top} C[cap top S] ==> C[S]
-- control commands to compute the diagrams
"forking_diagrams"   <- overlap (top).l all
"commuting_diagrams" <- overlap (top).r all
-- calling
--  lrsx join simple.inp
--  lrsx induct atp-path=aprove/ forking_diagrams
--  lrsx induct atp-path=aprove/ commuting_diagrams
-- will generate the diagrams and perform the automated induction
-- (it is assumed that aprove.jar and ceta are in the path specified by atp-path)
```