Automating the Diagram Method to Prove Correctness of Program Transformations

David Sabel

Automating the Diagram Method to Prove Correctness of Program Transformations

David Sabel

Goethe-University
Frankfurt am Main, Germany
`sabel@ki.cs.uni-frankfurt.de` *

Our recently developed LRSX Tool implements a technique to automatically 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 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. [21]) 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 [3, 2, 18]).

The LRSX Tool[1] supports correctness proofs of program transformations in those calculi by automating the so-called diagram method (see e.g. [18, 14] and also [8, 20]) 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. [4]), 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 (where the diagrams are represented in an abstract manner, i.e. only the names and directions of

---

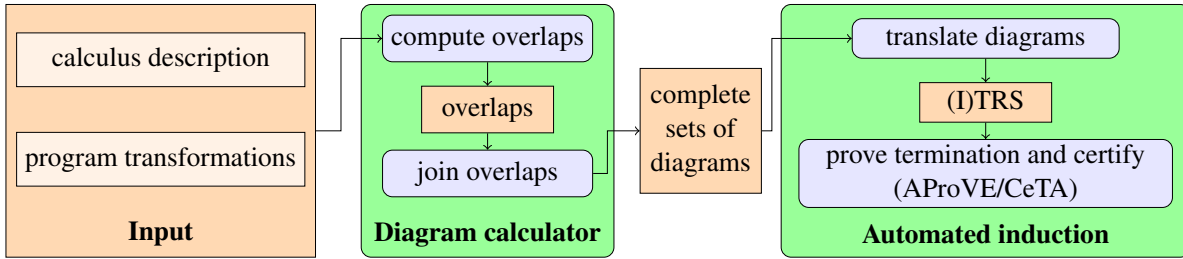[1]available from http://goethe.link/LRSXTOOL61

Figure 1: The overall structure of the automated diagram method

the reduction and transformation steps are kept, but all information on the meta-expressions is removed) such that termination of the system implies correctness of the program transformations. The automated termination prover AProVE [1, 7] and the certifier CeTA [5, 19] are used to automate these steps. We will explain core components of the automated method and illustrate the use of the LRSX Tool.

As a running example, we use the call-by-need lambda calculus with letrec L_{need} [17]. Its syntax, small-step operational semantics (called standard reduction), and the program transformations (gc1) and (gc2) to perform garbage collection, and transformations (cp-in) and (cp-e) to copy abstractions, are shown in Fig. 2. Standard reduction implements the lazy evaluation strategy with sharing by applying small-step reduction rules at needed positions, which are determined by application contexts, reduction contexts, and chains of letrec-bindings that occur as variable-to-variable bindings and also as chains $\{w_i = A[w_{i+1}]\}_{i=1}^m$. Reduction is meant modulo (extended) α-renaming, i.e. α-equivalent expressions where letrec-bindings are treated like a set are not distinguished.

Outline. In Sect. 2 we explain the meta language and the input for the diagram method. In Sect. 3 we describe the automated correctness proof for the standard cases, and in Sect. 4 we discuss extensions which are also built in the tool. In Sect. 5 we report on some experiments and we conclude in Sect. 6.

2 Program Calculi and Transformations

The input of the diagram technique is a program calculus – consisting of definitions of contexts, standard reduction rules, answers representing successfully evaluated programs – and a set of program transformations. Rules and answers are expressed in the meta-language LRSX (see also [16]). This meta-language is parametrized over a set $\mathscr{F}$ of function symbols and a finite set $\overline{K}$ of context classes [2]. The *syntax of* LRSX-*expressions* $\mathbf{Exp} = \mathbf{HExp}^0$, a countably-infinite set of variables $\mathbf{Var}$, higher-order expressions of order n $\mathbf{HExp}^n$, environments $\mathbf{Env}$, and bindings $\mathbf{Bind}$ is defined by the grammar

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

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 $\mathbf{Var}$, or $\mathbf{HExp}^{k_i}$. We assume $\{\texttt{var}, \lambda\} \subseteq \mathscr{F}$ where 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 $\mathbf{Var}$ are denoted by $\times$,

---

[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.

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

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

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

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

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

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

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 for all $i : w_i$ does not occur in Env, e

(gc2) $\texttt{letrec } w_1 {=} e_1, \ldots, w_n {=} e_n \texttt{ in } e \to e$, if for all $i : w_i$ does 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 2: The calculus L_{need}: Weak head normal forms (WHNFs) are $\lambda w.e$ or $\texttt{letrec } Env \texttt{ in } \lambda w.e$

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) that occurs with a **Var**-argument x, and an **Exp**-argument s. Each context variable D has a class $cl(D)$ and each *Ch*-variable has a class $cl(Ch)$. An LRSX-expression s is *ground* (written as s) iff it does not contain any meta-variable.

Example 2.1. The syntax of the λ-calculus (and also of our running example L_{need}) can be expressed in LRSX, by the function symbols var, λ, and 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 $(\lambda(\text{x.var x})) \ (\lambda(\text{x.var x}))$. 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 $(\text{Nil} \to e_1) \ (\text{Cons x xs} \to e_2)$ – to deconstruct lists can be represented as caselist $l \ e_1 \ \text{x.xs}.e_2$ where caselist is a function symbol of type **Exp** $\to$ **Exp** $\to$ **HExp**^{2} $\to$ **Exp**.

Contexts are expressions, where the hole $[\cdot]$ occurs instead of one subexpression. With d we denote a ground context and d denotes LRSX-contexts, i.e. 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 $[\cdot_1], \ldots, [\cdot_k]$. A *context class* $\mathcal{K} \in \overline{K}$ is a set of contexts which is defined by a context free grammar describing the syntax of contexts and by a *prefix* and a *forking table*, which 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 $(\mathcal{K}_1, \mathcal{K}_2)$ to a pair of classes $(\mathcal{K}_3, \mathcal{K}_4)$ such that for context variables

```
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 3: Definition of application and top-contexts as input for the LRSX Tool

D_i with $cl(D_i) = \mathcal{K}_i$ 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 ($\mathcal{K}_1$, $\mathcal{K}_2$) to a set of tuples of the form ($\mathcal{K}_3$, $\mathcal{K}_4$,$\mathcal{K}_5$,$d[\cdot_1, \cdot_2]$) such that for context variables D_i of class $\mathcal{K}_i$ 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 $D_1 \mapsto D_3[d[D_4[\cdot], D_5[t]]], D_2 \mapsto D_3[d[D_4[s], D_5[\cdot]]]$.

For calculus L_{need}, it suffices to 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. 3. 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 $(A, A, T, \text{app } [\cdot_1] [\cdot_2])$, 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 pathes go through the enviroment, 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 $cl(Ch) = \mathcal{K}$ stands for $x.\mathsf{d}[s]$ or chains $x.\mathsf{d}_1[(\text{var } x_1)];x_1.\mathsf{d}_2[(\text{var } x_2)];\dots;x_n.\mathsf{d}_n[s]$ with fresh x_i and contexts d, d_i of class $\mathcal{K}$. 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 d, $CV(\mathsf{d})$ (the *captured variables*) is the set of variables x which become bound if plugged into the hole of 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.

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].

Definition 2.2. For $\ell, r \in \mathbf{Exp}$, a constraint tuple Δ, $\kappa \in \{SR, \mathsf{T}\}$, a name *n*, $\ell \xrightarrow{\kappa, n}_\Delta r$ is called a *letrec rewrite rule*, provided that $MV(\Delta) \subseteq MV(\ell) \cup MV(r)$ and in each of the expressions ℓ and *r*, every variable of type *S* occurs at most twice; every variable of kind *E*, *Ch*, *D* occurs at most once; if $\kappa = SR$, then *Ch*-variables occurring in ℓ must occur in one `letrec`-environment only; 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 $\gamma(\ell \xrightarrow{\kappa, n}_\Delta r) := \left\{ \rho(\ell) \to \rho(r) \,\middle|\, \begin{array}{l} \rho \text{ is a ground substitution for } \ell, r, \\ \text{the LVC holds for } \rho(\ell), \rho(r), \rho \text{ satisfies } \Delta \end{array} \right\}$. 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 $1 \leq i \leq m$. We write $\mathsf{s} \xrightarrow{\kappa, n_i}_\alpha \mathsf{s}'$ 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, and answers represent successful programs:

Definition 2.3. A letrec rewrite rule $\ell \xrightarrow{\kappa, n}_\Delta r$ is a *standard reduction* if $\kappa = SR$ and: 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.

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. 4, where several rule variants are used for the different cases of a reduction context and side conditions of the rules (see Fig. 2) are expressed by constraints. For instance, rule $(SR, lbeta)$ is "implemented" by three rules {SR, lbeta, 1}, {SR, lbeta, 2}, and {SR, lbeta, 3}: one variant for each variant of the reduction contexts *R*, and the rule {SR, llet-in, 1} requires a NCC to ensure that let-bound variables in E2 are disjoint from the variables in E1. Answers in L_{need} are the weak head normal forms, i.e. abstractions perhaps with an outer `letrec`.

---

[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 (`letrec` *env* `in` *c*, *d*) for some constant *c*.

[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,cp-in,1}    letrec VC|Xn,\X.S|; E in A[var Xn] ==> letrec VC|Xn,\X.S|; E in A[\X.S]
{SR,cp-e,1}     letrec VC|Xn,\X.S|; Ch^A[Y,A1[var Xn]];E in A[var Y]
                  ==> letrec  VC|Xn,\X.S|; Ch^A[Y,A1[\X.S]]; E in A[var Y]
                  where A1 /= [.]
{SR,llet-in,1}  letrec E1 in letrec E2 in S ==> letrec E1;E2 in S
                  where E1 /= {}, E2 /= {}, [E1,letrec E2 in [.]]
{SR,llet-e,1}   letrec E1;X=letrec E2 in S in A[var X] ==> letrec E1;E2;X=S in A[var X]
                  where E2 /= {},[E1,letrec E2 in [.]],(A[var X],letrec E2 in [.])
{SR,llet-e,2}   letrec E1;X=letrec E2 in S;Ch^A[Y,var X] in A[var Y]
                  ==> letrec E1;E2;X=S;Ch^A[Y,var X] in A[var Y]
                  where E2 /= {},[E1;Ch^A[Y,var X],letrec E2 in [.]],
                        (var X,letrec E1;E2;Ch^A[Y,var X] in [.]),
                        (A[var Y],letrec E2 in [.])
{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 /= {}
```

The notation Ch^A means that chain variable Ch is of context class A, and VC|.,.| denotes the chain variable VC of context class *Triv* which is the (built-in) class containing only the empty context.

Figure 4: Standard reductions and answers for L_{need} as input for the LRSX Tool

Definition 2.4. For a program calculus (SR, Ans), a ground expression s_0, s_0 *converges* (written $s_0\downarrow$) iff there exists a sequence $s_0 \xrightarrow[\alpha]{SR} s_1 \xrightarrow[\alpha]{SR} \dots \xrightarrow[\alpha]{SR} s_k$ 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 d we have $d[s] \leq_\downarrow d[t]$, 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:

Definition 2.5. A letrec rewrite rule with $\kappa = T$ is a *meta transformation*, if the following conditions hold (see also Fig. 5): For all s_1, s_2, t_1 with $s_1 \xrightarrow{T,n} s_2$, $s_1 \sim_\alpha t_1$, such that t_1 fulfills the DVC: 1. If $t_1 \in \gamma(t, \Delta)$ for some $(t, \Delta) \in Ans$, then there exists $s_1' \in \gamma(t, \Delta)$ such that $s_1' \sim_\alpha s_1$ and $s_1' \xrightarrow{T,n} s_2'$ 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 $s_1' \xrightarrow{T,n} s_2'$, and $s_1' \xrightarrow{SR,n'} t_2'$.

A meta transformation $\ell \xrightarrow{T,n}_\Delta r$ is *correct* iff $\gamma(\ell \xrightarrow{T,n}_\Delta r) \subseteq \sim_c$. A meta transformation $\ell \xrightarrow{T,n}_\Delta r$ 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. $s \xrightarrow{T,n} t$, $s \leq_\downarrow t$ imply $s \leq_c t$.

The conditions on meta transformations allow us to inspect overlaps between transformations and standard reductions and answers without considering α-renaming steps. A sufficient criterion to fulfill
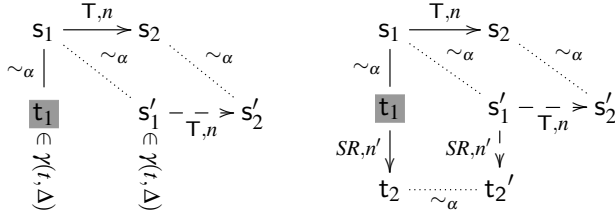
Figure 5: Illustration of Cond. 1 and 2 in Def. 2.5: solid lines are given relations, dotted / dashed lines are existentially quantified relations, t_1 fulfills the DVC.
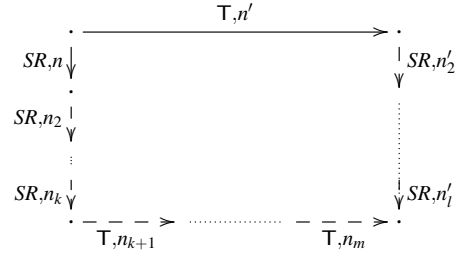

Figure 6: Representation of a forking diagram

Conditions (1) and (2) from Definition 2.5 is that applicabilty of a transformation to an expression *s* implies applicabilty of the transformation to all α-renamed expressions $s' \sim_\alpha s$ that fulfill the DVC (see Proposition A.1 in the appendix). In the calculus L_{need}, this conditions holds for most of the transformations under consideration. An exception is the reversed copy transformation, (e.g. the reversal of $\xrightarrow{T,cp\text{-}in}$ in Fig. 2). The rule does not fulfill the mentioned condition, since all ground instances of the left hand side violate the DVC. However, the conditions (1) and (2) from Definition 2.5 hold, since two occurrences of $\lambda v.e$ do not forbid the application of a standard reduction.

Meta transformations $\ell \xrightarrow{T,n}_\Delta r$ 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 [15] 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 [.])
```

3   Computing Diagrams and Automated Induction

For proving $\gamma(gcT) \subseteq \leq_\downarrow$, we have to compute all overlaps between an answer and the left hand side of (gcT) (called *answer overlaps*[5]) and all overlaps between the left hand sides of a standard reduction and of (gcT) (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 [16] 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 and computing so-called *answer diagrams* and *forking diagrams* (consisting of the overlap and a join), 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 $s \xrightarrow{T,n'} t$, 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 $s_1 \xleftarrow{SR,n} t \xrightarrow{T,n'} t_1$, a *join* is a sequence $s_1 \xrightarrow{SR,n_2}_\alpha \cdots \xrightarrow{SR,n_k}_\alpha s_k \xrightarrow{T,n_{k+1}}_\alpha$

---

[5]Internally computing answer overlaps is done by adding special standard reduction rules of the form $\ell \to answer$ where $\ell \in \mathsf{Ans}$ and *answer* is a new constant.

[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`.
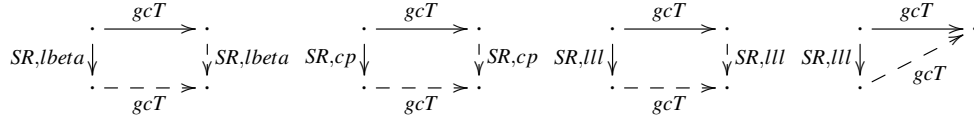
Figure 7: Diagrams for (gcT), pictorial

```
<-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-
```

```
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 8: Diagrams for (gcT), textual   Figure 9: Obtained TRS for (gcT)

$\cdots \xrightarrow{\mathsf{T},n_m}_{\alpha} s_m \sim_{\alpha} t_l \xleftarrow{\mathsf{SR},n'_l}_{\alpha} \cdots \xleftarrow{\mathsf{SR},n'_2}_{\alpha} t_1$ 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. 6 (where steps from the overlap are written with solid arrows, and (existentially quantified) steps of the join are written with dashed arrows) .

To apply the letrec rewrite rules a matching algorithm for LRSX is used which is described in [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, and thus the algorithm has to guarantee that the given constraints imply the needed constraints before delivering a matcher. A further specialty is that 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. The LRSX 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.

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. In Appendix B a sound check for proving equivalence of constrained expressions is provided.

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. 8 and a pictorial representation of the forking diagrams is in Fig. 7. In a pen-and-paper proof of $\gamma(gcT) \subseteq \leq_{\downarrow}$, an induction on the length of a converging reduction sequence $s \xrightarrow{\mathsf{SR},*} s'$ 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 $s \xrightarrow{\mathsf{SR}} s_1 \xrightarrow{\mathsf{SR},*} s'$. Applying a forking diagram to $s_1 \xleftarrow{\mathsf{SR}} s \xrightarrow{gcT} t$ shows existence of some t' with $s_1 \xrightarrow{gcT} t' \xleftarrow{\mathsf{SR}} t$ or $s_1 \xrightarrow{gcT} t' = t$ and by the induction hypothesis $t'\downarrow$ which also implies $t\downarrow$. This induction
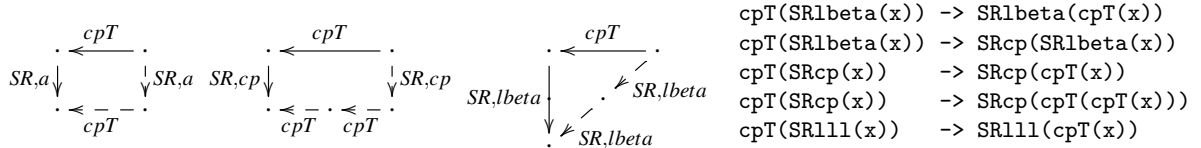
---

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

Figure 10: 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 11: A term rewrite rule for the second diagram

(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 show in Fig. 9 which is shown to be innermost terminating using the prover AProVE and the certifier CeTA.

4   Extended Techniques and Limitations of the Method

The previous example (proving $\gamma(gcT) \subseteq \leq_\downarrow$) is quite simple where 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

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

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. 10.

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

|	# 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

free variables on the right hand sides (or alternatively integer term rewrite systems, see [11, 6]) where 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. 11. 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. 2 closed by top-contexts. Computing commuting diagrams results in the following diagrams and the corresponding term rewrite system:


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

The TRS is non-terminating. If we split the transformation (cpT) into transformations where the copy target is a top-context (tcpT) , and another transformation (dcpT) where the target is below an abstraction, then the diagram set becomes
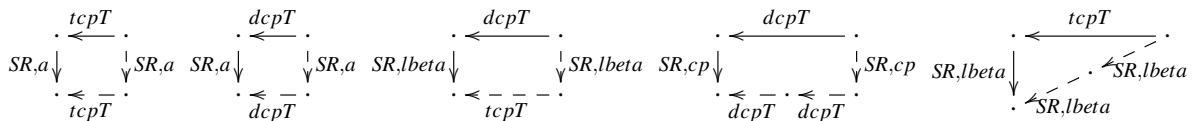
and termination of the corresponding TRS can be proved.

5 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 are the calculus L_{need} [17] – 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 [18] 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.

The results of our experiments 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. The time to compute (more) joins in the calculus LR for commuting diagrams than for computing forking diagrams, can be explained: we optimized the diagram computation (but cutting down unusual search paths) for the commuting case much more than for the forking case.

6 Conclusion

We presented a system to automatically prove correctness of program transformations which is implemented as the LRSX Tool. 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. Future work is to extend the tool to prove correctness of program translations where source and target language are different.

Acknowledgments. We thank René Thiemann for support on AProVE and CeTA.

References

[1] AProVE (2018): *Homepage of AProVE.* http://aprove.informatik.rwth-aachen.de.

[2] Zena M. Ariola & Matthias Felleisen (1997): *The Call-By-Need lambda Calculus.* J. Funct. Program. 7(3), pp. 265–301.

[3] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky & Philip Wadler (1995): *A call-by-need lambda calculus.* In: *POPL 1995*, ACM, pp. 233–246.

[4] Franz Baader & Tobias Nipkow (1998): *Term Rewriting and All That.* Cambridge University Press.

[5] CeTA (2018): *Homepage of CeTA.* http://cl-informatik.uibk.ac.at/software/ceta.

[6] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp & Stephan Falke (2009): *Proving Termination of Integer Term Rewriting.* In: *RTA 2009*, LNCS 5595, Springer, pp. 32–47.

[7] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski & René Thiemann (2014): *Proving Termination of Programs Automatically with AProVE.* In: *IJCAR 2014*, LNCS 8562, Springer, pp. 184–191.

[8] Elena Machkasova & Franklyn A. Turbak (2000): *A Calculus for Link-Time Compilation.* In: *ESOP 2000*, LNCS 1782, Springer, pp. 260–274.

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

[10] Gordon D. Plotkin (1975): *Call-by-name, call-by-value, and the lambda-calculus.* Theoret. Comput. Sci. 1, pp. 125–159.

---

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

[11] Conrad Rau, David Sabel & Manfred Schmidt-Schauß (2012): *Correctness of Program Transformations as a Termination Problem*. In: *IJCAR 2012*, *LNCS* 7364, Springer, pp. 462–476.

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

[13] David Sabel (2017): *Matching of Meta-Expressions with Recursive Bindings*. In: *Informal Proceedings of UNIF 2017*.

[14] David Sabel & Manfred 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.

[15] Manfred Schmidt-Schauß & David Sabel (2010): *On generic context lemmas for higher-order calculi with sharing*. *Theoret. Comput. Sci.* 411(11-13), pp. 1521 – 1541.

[16] Manfred Schmidt-Schauß & David Sabel (2016): *Unification of Program Expressions with Recursive Bindings*. In: *PPDP 2016*, ACM, pp. 160–173.

[17] Manfred Schmidt-Schauß, David Sabel & Elena Machkasova (2010): *Simulation in the Call-by-Need Lambda-Calculus with letrec*. In: *RTA 2010*, *LIPIcs* 6, Schloss Dagstuhl, pp. 295–310.

[18] Manfred Schmidt-Schauß, Marko Schütz & David Sabel (2008): *Safety of Nöcker's Strictness Analysis*. *J. Funct. Programming* 18(04), pp. 503–551.

[19] René Thiemann & Christian Sternagel (2009): *Certification of Termination Proofs Using CeTA*. In: *TPHOLs 2009*, *LNCS* 5674, Springer, pp. 452–468.

[20] Joe B. Wells, Detlef Plump & Fairouz Kamareddine (2003): *Diagrams for Meaning Preservation*. In: *RTA 2003*, *LNCS* 2706, Springer, pp. 88 –106.

[21] Andrew K. Wright & Matthias Felleisen (1994): *A Syntactic Approach to Type Soundness*. *Inf. Comput.* 115(1), pp. 38–94.

Appendix

A   Soundness of the Diagram Method

Proposition A.1. *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.*

We show soundness of the diagram method, where it is important to verify that our conditions in Definition 2.5 imply that it suffices to consider overlaps of standard reductions and transformations without performing α-renaming.

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, a diagram in the set is applicable. 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.

Already, in [11] it was shown that proving termination of the string rewrite system with infinitely many rules can be automated by using automated termination provers for term rewrite systems and showing termination of the integer term rewrite system (or a term rewrite system with free variables on the right hand side that represent arbitrary construtor terms). Thus, 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.2. *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 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 $\mathsf{s}_1 \xleftarrow{SR}_\alpha \mathsf{s}_2 \xrightarrow{SR}_\alpha \mathsf{s}_3$, then the calculus is deterministic and thus $\mathsf{s}_1 \sim_\alpha \mathsf{s}_3$ holds. Replace the $\mathsf{s}_1 \xleftarrow{SR}_\alpha \mathsf{s}_2 \xrightarrow{SR}_\alpha \mathsf{s}_3$ by $\mathsf{s}_1 \sim_\alpha \mathsf{s}_3$.

4. If the sequence has a prefix $s_1 \xrightarrow{SR}_\alpha s_3$ where s_1 is an answer, then the calculus is deterministic and s_3 is answer and we replace the prefix $s_1 \xrightarrow{SR}_\alpha s_3$ by s_3.

5. Subsequences $s_1 \sim_\alpha s_2 \sim_\alpha s_3$ are replaced by $s_1 \sim_\alpha s_3$.

6. If the left-most expression of the sequence is $s_1 \in \gamma(\mathsf{Ans})$ and does not fulfill the DVC, then replace s_1 by $s_1' \sim_\alpha s_1$ such that s_1' fulfills the DVC. Note that due to our assumption on answers, $s_1' \in \gamma(\mathsf{Ans})$.

7. If the sequence has a prefix $t_1 \sim_\alpha s_1 \xrightarrow{T,n'} s_2$, where t_1 fulfills the DVC and $t_1 \in \gamma(\mathsf{Ans})$, then first apply Condition (1) of Definition 2.5, i.e. replace the prefix by $t_1 \sim_\alpha s_1' \xrightarrow{T,n} s_2' \sim_\alpha s_2$ where $s_1' \in \gamma(\mathsf{Ans})$ and $s_1' \sim_\alpha t$. Now the answer overlap $s_1' \xrightarrow{T,n} s_2'$ is replaced by the corresponding join.

8. If the sequence contains $t_2 \xleftarrow{SR,n'} t_1 \sim_\alpha s_1 \xrightarrow{T,n''} s_2$, then t_1 fulfills the DVC (by the modification in item 2) and we can use Condition 2 of Definition 2.5 and replace $t_2 \xleftarrow{SR,n'} t_1 \sim_\alpha s_1 \xrightarrow{T,n''} s_2$ by $t_2 \sim_\alpha t_2' \xleftarrow{SR,n'} s_1' \xrightarrow{T,n''} s_2' \sim_\alpha s_2$. Now replace the forking overlap $t_2' \xleftarrow{SR,n'} s_1' \xleftarrow{T,n''} s_2'$ 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 $\xrightarrow{SR}_\alpha$-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   Checking Equivalence of Constrained Expressions

We define an NCC-implication check for constrained expressions. Before providing the implication check, we define how to split NCCs into *atomic NCCs* (u,v) such that u,v are variables or meta-variables. For a set $\mathscr{S}$ of NCCs, let $split_{ncc}(\mathscr{S}) := \bigcup_{(s,d)\in\mathscr{S}} Var_M(s) \times CV_M(d)$ where $Var_M(s) = MV(s) \cup Var(s)$, and CV_M collects all concrete variables that capture variables of the context hole, and all meta-variables which may have concretizations that introduce capture variables. A ground substitution ρ *satisfies an atomic NCC* (u,v) iff $Var(\rho(u)) \cap CV_A(\rho(v)) = \emptyset$ where $CV_A(\mathsf{x}) = \{\mathsf{x}\}$ for all variables x and $CV_A(r) = CV(r)$ for all other constructs r. It is easy to verify that for a set of NCCs $\mathscr{S}$, ρ satisfies all constraints in $\mathscr{S}$ if and only if ρ satisfies all atomic NCCs in $split_{ncc}(\mathscr{S})$.

We now define the NCC-implication check.

Definition B.1. Let (s,∇) and (t,Δ) be constrained expressions with $s \sim_{let} t$, $\gamma(s,\nabla) \neq \emptyset, \gamma(t,\Delta) \neq \emptyset$ and $MV(s) = MV(t)$. Then ∇ *implies* Δ iff i) for all $D \in \Delta_1 : D \in \nabla_1$, ii) for all $E \in \Delta_2$: $E \in \nabla_2$ and iii) for all $(u,v) \in split_{ncc}(\Delta_3) \cup NCC_{dvc}(t)$ one of the following cases holds:

1. $u = \mathsf{x}$ and $v = \mathsf{y}$ where $\mathsf{x} \neq \mathsf{y}$.

2. $(u,v) \in split_{ncc}(\nabla_3) \cup NCC_{dvc}(s)$.

3. $u = v, u \in MV(\Delta) \backslash MV(\nabla)$ and $u \in \{Ch,D,E\}$ such that $E \notin \Delta_2$.

4. $u \neq v, u \in MV(\Delta) \backslash MV(\nabla)$ and $u = Ch$, or $u = S$, or $u = D$ or $u = E$, or $u = X$.

5. $u \neq v, v \in MV(\Delta) \backslash MV(\nabla)$ and $v \in \{Ch,D,E,X\}$.

6. $u = E$ or $u = Ch$ with $cl(Ch) = Triv$, and $(u,u) \in split_{ncc}(\nabla_3) \cup NCC_{dvc}(s)$.

7. $v \in \{E, Ch, D\}$ and $(v,v) \in split_{ncc}(\nabla_3) \cup NCC_{dvc}(s)$.

8. (u,v) is of the form (X,y), (x,Y), (X,Y), (x,D), (X,D), (x,E), (X,E), (x,Ch), (X,Ch), (Ch_1,x), (Ch_1,X), (Ch_1,E), (Ch_1,D), or (Ch_1,Ch_2) where $cl(Ch_1) = Triv$ and in all cases $(v,u) \in split_{ncc}(\nabla_3) \cup NCC_{dvc}(s)$.

We call (s,∇) and (t,Δ) *NCC-equivalent* iff Δ implies ∇ and ∇ implies Δ.

Proposition B.2. *NCC-equivalence of (s,∇) and (t,Δ) implies $\gamma(s,\nabla) = \gamma(t,\Delta)$.*

Proof. It suffices to show that for (s,∇) and (t,Δ) such that ∇ implies Δ, the inclusion $\gamma(t,\Delta) \subseteq \gamma(s,\nabla)$ holds. Let ρ be a substitution such that $\rho(s)$ satisfies the LVC and ρ satisfies ∇. We show that there exists a substitution ρ_0 such that $\rho_0 \circ \rho$ is ground for (t,Δ) and with $\rho' = \rho_0 \circ \rho$, $\rho'(t)$ satisfies the LVC and ρ' satisfies Δ.

Let $\rho_0(Ch) = [\cdot_1].[\cdot_2]$ for all $Ch \in MV(t,\Delta) \setminus MV(s,\nabla)$ $\rho_0(S) = \lambda \mathsf{x}_S.\mathsf{x}_S$ for a fresh variable x_S for all $S \in MV(t,\Delta) \setminus MV(s,\nabla)$. For all $D \in MV(t,\Delta) \setminus MV(s,\nabla)$, let $\rho_0(D) = [\cdot]$ if $D \notin \Delta_1$, and $\rho_0(D) = \mathsf{d}$ where d is a context with $CV(\mathsf{d}) = \emptyset$. For all $E \in MV(t,\Delta) \setminus MV(s,\nabla)$, let $\rho_0(E) = \emptyset$ if $E \notin \Delta_2$ and $\rho_0(E) = \mathsf{x}_E.\mathtt{var}\ \mathsf{x}_E$, otherwise where x_E is a fresh variable; For all $X \in MV(t,\Delta) \setminus MV(s,\nabla)$, let $\rho_0(X) = \mathsf{x}_X$ for a fresh variable x_X.

By the definition of the NCC-implication check and the choice of ρ_0, ρ' satisfies Δ_1 and Δ_2. For the remaining parts, we show that ρ' satisfies all atomic NCCs in $split_{ncc}(\Delta_3) \cup NCC_{dvc}(t)$.

Let $(u,v) \in split_{ncc}(\Delta_3) \cup NCC_{dvc}(t)$. Then one of the cases of Definition B.1 holds. If $u = \mathsf{x}$ and $v = \mathsf{y}$ where $\mathsf{x} \neq \mathsf{y}$, the $(\rho(u), \rho(v))$ is satisfied. If $(u,v) \in split_{ncc}(\nabla_3) \cup NCC_{dvc}(s)$ then ρ' satisfies (u,v) since ρ satisfies ∇_3 or for $(u,v) \in NCC_{dvc}(s)$, u,v are let-variables of the same environment and thus ρ must map u and v to distinct concrete variables, since otherwise the LVC for $\rho(s)$ is violated. If $u = v$, $u \in MV(\Delta) \setminus MV(\nabla)$ and $u \in \{Ch,D,E\}$ such that $E \notin \Delta_2$ we verify that ρ_0 satisfies (u,u) and thus ρ does. If $u \neq v$, $u \in MV(\Delta) \setminus MV(\nabla)$ and $u=Ch$, or $u=S$, or $u=D$ or $u=E$, or $u=X$. ρ_0 satisfies (u,u) and thus ρ does. If $u \neq v$, $v \in MV(\Delta) \setminus MV(\nabla)$ and $v \in \{Ch,D,E,X\}$. ρ_0 satisfies (u,u) and thus ρ does. If (u,v) is (X,y), (x,Y), (X,Y), (x,D), (X,D), (x,E), (X,E), (x,Ch), (X,Ch), (Ch_1,x), (Ch_1,X), (Ch_1,E), (Ch_1,D), or (Ch_1,Ch_2) where $cl(Ch_1) = Triv$ and in all cases $(v,u) \in split_{ncc}(\nabla_3) \cup NCC_{dvc}(s)$, we verify that since ρ satisfies (v,u), ρ also has to satisfy (u,v). $\square$