Automating the Diagram Method to Prove Correctness of Program Transformations

David Sabel†

Goethe-University Frankfurt am Main, Germany

WPTE 2018, July 8th, Oxford, UK

†Research supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA 2908/3-1.
Motivation

- reasoning on program transformations w.r.t. operational semantics
- for program calculi with higher-order constructs and recursive bindings, e.g. letrec-expressions:

  \[
  \text{letrec } x_1 = s_1; \ldots; x_n = s_n \text{ in } t
  \]

- extended call-by-need lambda calculi with letrec that model core languages of lazy functional programming languages like Haskell
Correctness of Program Transformations

A program transformation $T$ is a binary relation on expressions. It is correct iff $e \xrightarrow{T} e' \implies (\forall \text{contexts } C : C[e] \downarrow \iff C[e'] \downarrow)$

$\downarrow$ means successful evaluation $e \downarrow := e \xrightarrow{sr,*} e'$ and $e'$ is a successful result

- where $\xrightarrow{sr}$ is the small-step operational semantics (standard reduction)
- and $\xrightarrow{sr,*}$ is the reflexive-transitive closure of $\xrightarrow{sr}$

As a core proof method, we need to show

**convergence preservation:** $e \xrightarrow{T'} e' \implies (e \downarrow \iff e' \downarrow)$

where $T'$ is a contextual closure of $T$
Idea of the Diagram Method

- **Base case:** For all successful $e$
  - $e \rightarrow e'$
  -successful

- **General case:** For all programs $e$
  - $e \rightarrow e'$
  -standard reduction steps
  - $e'' \rightarrow e'''$
  -program transformation steps

- **Inductive construction**
  - $e \rightarrow e'$
  -standard reduction steps
  - $e'' \rightarrow e'''$
  -by the induction hypothesis
  - $\ldots$
  -$e_4 \rightarrow e_5$
  -successful
Focused Languages and Previous Results

The diagram technique was, for instance, used for

- **call-by-need** lambda calculi with `letrec`, data constructors, case, and `seq` [SSSS08, JFP] and **non-determinism** [SSS08, MSCS]
- **process calculi** with call-by-value [NSSSS07, MFPS] or call-by-need evaluation [SSS11, PPDP] and [SSS12, LICS]
- reasoning on whether program transformations are **improvements** w.r.t. the **run-time** [SSS15, PPDP], [SSS17, SCP], [SSSD18, PPDP] and **space** [SSD18, WPTE]
Focused Languages and Previous Results

The diagram technique was, for instance, used for

- **call-by-need** lambda calculi with `letrec`, data constructors, case, and seq [SSSS08, JFP] and non-determinism [SSS08, MSCS]
- **process calculi** with call-by-value [NSSSS07, MFPS] or call-by-need evaluation [SSS11, PPDP] and [SSS12, LICS]
- reasoning on whether program transformations are improvements w.r.t. the **run-time** [SSS15, PPDP], [SSS17, SCP], [SSSD18, PPDP] and **space** [SSD18, WPTE]

**Conclusions from these works**

- The diagram method works well
- The method requires to compute overlaps (error-prone, tedious,...)
- Automation of the method would be valuable
Automation of the Diagram-Method

Structure of the LRSX-Tool

Input
- calculus description
- program transformations

Diagram calculator
- compute overlaps
- overlaps
- join overlaps

Automated induction
- translate diagrams
- (I)TRS
- prove termination (AProVE/CeTA)

6/22
Representation of the Input

Structure of the LRSX-Tool
Requirements on the Meta-Syntax

Operational semantics of typical call-by-need calculi (excerpt)

Reduction contexts:
\[ A ::= [\cdot] | (A e) \]
\[ R ::= A | \text{letrec } Env \text{ in } A | \text{letrec } \{ x_i = A_i[x_{i+1}] \}_{i=1}^{n-1}, x_n = A_n, Env, \text{ in } A[x_1] \]

Standard-reduction rules and some program transformations:

(SR,lbeta) \[ R[(\lambda x.e_1) e_2] \to R[\text{letrec } x = e_2 \text{ in } e_1] \]

\[ \ldots \]

(T,cpx) \[ T[\text{letrec } x = y, Env \text{ in } C[x]] \to T[\text{letrec } x = y, Env \text{ in } C[y]] \]

(T,gc,1) \[ T[\text{letrec } Env, Env' \text{ in } e] \to T[\text{letrec } Env' \text{ in } e], \]
\[ \text{if } \text{LetVars}(Env) \cap \text{FV}(e, Env') = \emptyset \]

(T,gc,2) \[ T[\text{letrec } Env \text{ in } e] \to T[e] \quad \text{if } \text{LetVars}(Env) \cap \text{FV}(e) = \emptyset \]

Meta-syntax must be capable to represent:

- contexts of different classes
- environments \( Env_i \) and environment chains \( \{ x_i = A_i[x_{i+1}] \}_{i=1}^{n-1} \)
Syntax of the Meta-Language LRSX

Variables
\[ x \in \text{Var} ::= X \quad \text{(variable meta-variable)} \]
\[ \quad \mid x \quad \text{(concrete variable)} \]

Expressions
\[ s \in \text{Expr} ::= S \quad \text{(expression meta-variable)} \]
\[ \quad \mid D[s] \quad \text{(context meta-variable)} \]
\[ \quad \mid \text{letrec } env \text{ in } s \quad \text{(letrec-expression)} \]
\[ \quad \mid \text{var } x \quad \text{(variable)} \]
\[ \quad \mid (f \ r_1 \ldots r_{ar}(f)) \quad \text{(function application)} \]
where \( r_i \) is \( o_i, s_i, \) or \( x_i \) specified by \( f \)

\[ o \in \text{HExpr}^n ::= x_1, \ldots, x_n.s \quad \text{(higher-order expression)} \]

Environments
\[ env \in \text{Env} ::= \emptyset \quad \text{(empty environment)} \]
\[ \quad \mid E; env \quad \text{(environment meta-variable)} \]
\[ \quad \mid \text{Ch}[x, s]; env \quad \text{(chain meta-variable)} \]
\[ \quad \mid x = s; env \quad \text{(binding)} \]

\( \text{Ch}[x, s] \) represents chains \( x=C_1[\text{var } x_1]; x_1=C_2[\text{var } x_2]; \ldots; x_n=C_n[s] \)
where \( C_i \) are contexts of class \( cl(Ch) \)
Binding and Scoping Constraints

There are restrictions on scoping and emptiness:

\[(T, cp) \quad T[\mathit{letrec} \ x = y, Env \ in \ C[x]] \to T[\mathit{letrec} \ x = y, Env \ in \ C[y]]\]
\[x, y \text{ are not captured by } C \text{ in } C[x], C[y]\]

\[(T, gc, 2) \quad T[\mathit{letrec} \ Env \ in \ e] \to T[e] \text{ if } Env \neq \emptyset, \ LetVars(Env) \cap FV(e) = \emptyset\]

We express them by constraint tuples \(\Delta = (\Delta_1, \Delta_2, \Delta_3)\):

- **non-empty context constraints** \(\Delta_1\): set of context variables
  - ground substitution \(\rho\) satisfies \(D \in \Delta_1\) iff \(\rho(D) \neq [\cdot]\)

- **non-empty environment constraints** \(\Delta_2\): set of environment variables
  - \(\rho\) satisfies \(E \in \Delta_2\) iff \(\rho(E) \neq \emptyset\)

- **non-capture constraints (NCCs)** \(\Delta_3\): set of pairs \((s, d)\)
  - \(\rho\) satisfies \((s, d)\) iff the hole of \(\rho(d)\) does not capture variables of \(\rho(s)\)
Representation of Rules

Standard reductions and transformations are represented as

$$\ell \rightarrow_\Delta r$$

where $\ell, r$ are LRSX-expressions and $\Delta$ is a constraint-tuple

Example:

$$(T, gc, 2) T[\text{letrec } Env \text{ in } e] \rightarrow T[e] \text{ if } \text{LetVars}(Env) \cap \text{FV}(e) = \emptyset$$

is represented as

$$D[\text{letrec } E \text{ in } S] \rightarrow (\emptyset, \{E\}, \{(S, \text{letrec } E \text{ in } [:])\}) D[S]$$
Computing Overlaps

Structure of the LRSX-Tool

- Input:
  - calculus description
  - program transformations

- Diagram calculator:
  - compute overlaps
  - overlaps
  - join overlaps

- Automated induction:
  - translate diagrams
  - (I)TRS
  - prove termination (AProVE/CeTA)
Computing Overlaps by Unification

\[ \sigma(\ell_A) = \sigma(\ell_B) \xrightarrow{\text{program transformation}} \sigma(r_B) \]

\[ \sigma(r_A) \xrightarrow{\text{unifier } \sigma \text{ for } \{\ell_A \vdash \ell_B\}} \]

Unification also has to respect the constraints \( \Delta_A \cup \Delta_B \).

Occurrence Restrictions:
S-variables at most twice, E-, Ch-, D-variables at most once

The Letrec Unification Problem is NP-complete \cite{SSS16, PPDP}
Algorithm UnifLRS \cite{SSS16, PPDP} is sound and complete and computes a finite representation of solutions.
Computing Overlaps by Unification

\[
\sigma(\ell_A) \quad = \quad \sigma(\ell_B) \quad \xrightarrow{\text{program transformation}} \quad \sigma(r_B)
\]

\[
\sigma(r_A) \quad \xrightarrow{\text{standard reduction}} \quad \sigma(r_A)
\]

unifier \( \sigma \) for \( \{ \ell_A \equiv \ell_B \}, \Delta_A \cup \Delta_B \)

- Unification also has to respect the constraints \( \Delta_A \cup \Delta_B \)
Computing Overlaps by Unification

\[
\sigma(\ell_A) = \sigma(\ell_B) \xrightarrow{\text{program transformation}} \sigma(r_B)
\]

\[
\sigma(r_A) \xrightarrow{\text{standard reduction}} \ast \xrightarrow{\text{unifier } \sigma \text{ for } (\{\ell_A \equiv \ell_B\}, \Delta_A \cup \Delta_B)}
\]

- Unification also has to respect the constraints \(\Delta_A \cup \Delta_B\)
- Occurrence Restrictions: \(S\)-variables at most twice, \(E\)-, \(Ch\)-, \(D\)-variables at most once
- The Letrec Unification Problem is NP-complete [SSS16, PPDP]
- Algorithm UnifLRS [SSS16, PPDP] is sound and complete
Computing Overlaps by Unification

\[ \sigma(\ell_A) = \sigma(\ell_B) \xrightarrow{\text{program transformation}} (\sigma(r_B), \Delta) \]

\[ (\sigma(r_A), \Delta) \xrightarrow{\text{standard reduction}} \]

output \((\sigma, \Delta)\) for \((\{\ell_A \doteq \ell_B\}, \Delta_A \cup \Delta_B)\)

- Unification also has to respect the constraints \(\Delta_A \cup \Delta_B\)
- Occurrence Restrictions: \(S\)-variables at most twice, \(E\)-, \(Ch\)-, \(D\)-variables at most once
- The Letrec Unification Problem is NP-complete [SSS16, PPDP]
- Algorithm UnifLRS [SSS16, PPDP] is sound and complete and computes a finite representation of solutions
Computing Joins

Structure of the LRSX-Tool

Input
- calculus description
- program transformations

Diagram calculator
- compute overlaps
- overlaps
- join overlaps

Automated induction
- translate diagrams
- (I)TRS
- prove termination (AProVE/CeTA)
Computing joins $\rightarrow^*$: abstract rewriting by rules $\ell \rightarrow^\Delta r$

- meta-variables in $\ell, r$ are instantiable and meta-variables in $t_i$ are fixed
- rewriting: match $\ell$ against $t_i$ and show that the given constraints $\nabla$ imply the needed constraints $\Delta$
- Sound and complete matching algorithm MatchLRS [Sab17, UNIF]
Example: (gc)-Transformation

\[(T,gc) := (T,gc,1) \cup (T,gc,2)\]

Unification computes 192 overlaps and joining results in 324 diagrams which can be represented by the diagrams:

\[
\begin{aligned}
&\cdot \rightarrow T,gc \rightarrow \cdot \\
&\downarrow SR,lbeta \rightarrow \cdot \\
&\cdot \rightarrow T,gc \rightarrow \cdot \\
&\downarrow SR,lbeta \\
&\cdot \rightarrow T,gc \rightarrow \cdot \\
&\downarrow SR,lll \\
&\cdot \rightarrow T,gc \rightarrow \cdot \\
&\downarrow SR,lll \\
&\cdot \rightarrow T,gc \\
&\downarrow SR,cp \\
&\cdot \rightarrow T,gc \rightarrow \cdot \\
&\downarrow SR,lbeta \\
&\cdot \rightarrow T,gc \rightarrow \cdot \\
&\downarrow SR,lbeta \\
&\cdot \rightarrow T,gc \\
&\downarrow SR,lll \\
&\cdot \rightarrow T,gc \rightarrow \cdot \\
&\downarrow SR,lll
\end{aligned}
\]

and the answer diagram

\[
\begin{aligned}
&Ans \rightarrow T,gc \rightarrow Ans \\
&Ans \rightarrow Ans
\end{aligned}
\]
Automated Induction

Structure of the LRSX-Tool

Input
- calculus description
- program transformations

Diagram calculator
- compute overlaps
- overlaps
- join overlaps

(A)TRS
- translate diagrams
- prove termination (AProVE/CeTA)
- Automated induction
Automated Induction: Ideas [RSSS12, IJCAR]

- Ignore the concrete expressions, only keep: kind of rule (SR or transformation) and rule-names, and answers as abstract constant

\[
\begin{align*}
    & T, gc \\
\end{align*}
\]

\[
\begin{align*}
    & SR, lbeta \downarrow \\
\end{align*}
\]

\[
\begin{align*}
    & T, gc \\
\end{align*}
\]

\[
\begin{align*}
    & T, gc \\
\end{align*}
\]

\[
\begin{align*}
    & Ans \xrightarrow{T, gc} Ans \\
\end{align*}
\]

Diagrams represent string rewrite rules on strings consisting of elements (SR, name), (T, name), and Answer (T, gc), (SR, lbeta) → (SR, lbeta), (T, gc) → (T, gc), Answer → Answer.

Termination of the string rewrite system implies successful induction. We use term rewrite systems and innermost-termination and apply AProVE and certifier CeTA.
Automated Induction: Ideas [RSSS12, IJCAR]

- Ignore the concrete expressions, only keep: kind of rule (SR or transformation) and rule-names, and answers as abstract constant

\[
\begin{align*}
T, gc & \quad \rightarrow \quad \cdot \\
SR, l beta & \downarrow \quad \rightarrow \quad \cdot \\
\cdot & \quad \rightarrow \quad \cdot \\
\cdot & \quad \rightarrow \quad \cdot \\
T, gc & \quad \rightarrow \quad \cdot \\
\end{align*}
\]

- Diagrams represent string rewrite rules on strings consisting of elements \((SR, name), (T, name), \) and Answer

\[
(T, gc), (SR, l beta) \rightarrow (SR, l beta), (T, gc) \quad \quad (T, gc), Answer \rightarrow Answer
\]
Automated Induction: Ideas [RSSS12, IJCAR]

- Ignore the concrete expressions, only keep: kind of rule (SR or transformation) and rule-names, and answers as abstract constant

\[
\begin{align*}
T, gc & \rightarrow \\
SR, lbeta \downarrow & \rightarrow \\
T, gc & \rightarrow \\
SR, lbeta \downarrow & \rightarrow \\
\end{align*}
\]

- Diagrams represent string rewrite rules on strings consisting of elements \((SR, name)\), \((T, name)\), and \(Answer\)

\[(T, gc), (SR, lbeta) \rightarrow (SR, lbeta), (T, gc) \quad (T, gc), Answer \rightarrow Answer\]

- Termination of the string rewrite system implies successful induction

- We use term rewrite systems and innermost-termination and apply AProVE and certifier CeTA
Advanced Techniques

Symbolic $\alpha$-Renaming

- Joining overlaps requires $\alpha$-renaming
  
  \[(\lambda X.S) (\text{letrec } E_1 \text{ in } S') \xrightarrow{T,gc} (\lambda X.S) (\text{letrec } E_1; E_2 \text{ in } S')\]
  
  \[
  \begin{aligned}
  &\text{sr, lbeta} \\
  &\text{letrec} \\
  &X = (\text{letrec } E_1 \text{ in } S') \quad \text{letrec } X = (\text{letrec } E_1; E_2 \text{ in } S') \text{ in } S \\
  \text{in } S \\
  \end{aligned}
  \]

  - $X = \text{may capture free occurrences of } X \text{ in } E_2!$

- **Solution:** Extend the meta-language and algorithms with **symbolic** $\alpha$-renamings [Sab17,PPDP]
Advanced Techniques (continued)

Transitive Closures

- Transitive closures of reduction / transformation rules, e.g.
  \[ A[\text{letrec } Env \text{ in } s] \xrightarrow{s_r,+} \text{letrec } Env \text{ in } A[s] \]
- Encoding of diagrams into TRSs uses free variables on right hand sides to “guess” the number of steps

Case-distinctions during search for joins

- Apply case distinctions whether environments \( E \) or contexts \( D \) are empty/non-empty and
- treat the cases separately

Rule reformulation (not automated)

- for a copy rule (cp) the diagram set is a nonterminating TRS
- Solution: \( cpT \): target of copy not below an abstraction
  \( cpd \): target of copy inside an abstraction
- The diagram set for \((cpT),(cpd)\) is a terminating TRS.
Experiments

- **LRSX Tool** available from [http://goethe.link/LRSXT00L61](http://goethe.link/LRSXT00L61)
  - computes diagrams and performs the automated induction

<table>
<thead>
<tr>
<th></th>
<th># overlaps</th>
<th># joins</th>
<th>computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Calculus $L_{need}$</strong> (11 SR rules, 16 transformations, 2 answers)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>2242</td>
<td>5425</td>
<td>48 secs.</td>
</tr>
<tr>
<td>$\leftarrow$</td>
<td>3001</td>
<td>7273</td>
<td>116 secs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th># overlaps</th>
<th># joins</th>
<th>computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Calculus $L^{+seq}_{need}$</strong> (17 SR rules, 18 transformations, 2 answers)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>4898</td>
<td>14729</td>
<td>149 secs.</td>
</tr>
<tr>
<td>$\leftarrow$</td>
<td>6437</td>
<td>18089</td>
<td>255 secs.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th># overlaps</th>
<th># joins</th>
<th>computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Calculus LR</strong> (76 SR rules, 43 transformations, 17 answers)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>87041</td>
<td>391264</td>
<td>$\sim$ 19 hours</td>
</tr>
<tr>
<td>$\leftarrow$</td>
<td>107333</td>
<td>429104</td>
<td>$\sim$ 16 hours</td>
</tr>
</tbody>
</table>
Conclusion and Outlook

Conclusion

- Automation of the diagram method for meta-language LRSX
- Algorithms for unification, matching, symbolic $\alpha$-renaming
- Encoding technique to apply termination provers for TRSs
- Experiments show: automation works well for call-by-need calculi

Further work

- Further calculi, e.g., process calculi with structural congruence
- Proving improvements
- Nominal techniques to ease reasoning on $\alpha$-renamings:
  - Nominal unification with letrec [SSKLV16, LOPSTR]
  - Nominal unification with context variables [SSS18, FSCD]
Thank you!