Correctness of Program Transformations: Automating Diagram-Based Proofs

David Sabel†

Goethe-University Frankfurt am Main, Germany

†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
A program transformation is a binary relation on program fragments

Before:
```c
for (X:=1, X < n, X++) {
    Z := Z + X;
}
```

After:
```c
X:=1
while X < n {
    Z := Z + X;
    X++; 
}
```
Motivation

A program transformation is a binary relation on program fragments

Some applications:
- Compilers: Optimizations (inlining, partial evaluation, . . .)
- Code Refactoring: Transformations to improve readability and maintainability
- Theorem Provers: Transforming programs in proofs
Correctness

Program transformation $T$ is correct iff $T \subseteq \sim_c$

- Contextual equivalence: $e \sim_c e'$ iff $e \leq_c e'$ and $e \geq_c e'$
- Contextual preorder: $e \leq_c e'$ iff $\forall C: C[e] \downarrow \implies C[e'] \downarrow$
- $\downarrow$ 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 $\xrightarrow{sr}$
Convergence Preservation

- **Convergence preservation:** $e \leq_{\downarrow} e' \iff e_{\downarrow} \implies e'_{\downarrow}$
- We only consider transformations $T$ such that $T \subseteq \leq_{\downarrow} \implies T \subseteq \leq_c$
- No restriction, since the contextual closure of $T$ fulfills this property.
- A context lemma allows for smaller closures (reduction contexts)
- $T \subseteq \geq_c$ can be proved by showing $T^{-1} \subseteq \leq_c$

Required task:
Idea of the Diagram Method

- Base case: For all successful $e$

\[ e \xrightarrow{\text{program transformation}} e' \]

successful
Idea of the Diagram Method

- Base case: For all successful $e$

```
\[ e \rightarrow e' \]
```

```
\text{successful} \quad \downarrow \quad \text{standard reduction steps} \quad \downarrow \quad \text{successful}
```

\[ e' \rightarrow e'' \]

\text{program transformation}

\[ e'' \text{ successful} \]
Idea of the Diagram Method

- **Base case:** For all successful \( e \)

\[
\begin{align*}
\text{program transformation} & \\
\downarrow & \\
\text{successful} & \\
\downarrow & \\
\text{standard reduction steps} & \\
\downarrow & \\
\text{successful} & \\
\end{align*}
\]

- **General case:** For all programs \( e \)

\[
\begin{align*}
\text{program transformation} & \\
\downarrow & \\
\text{standard reduction} & \\
\downarrow & \\
\text{successful} & \\
\end{align*}
\]
Idea of the Diagram Method

- **Base case:** For all successful $e$

  - $e$ \xrightarrow{\text{program transformation}} $e'$
  - $e'$ successful

  \[\vdash\text{standard reduction steps}\]

  \[\vdash e''\text{ successful}\]

- **General case:** For all programs $e$

  - $e$ \xrightarrow{\text{program transformation}} $e'$

  - $e'$ \xrightarrow{\text{standard reduction steps}} $e''$

  - $e''$ \xrightarrow{\cdots} $e'''$

- inductive construction
Idea of the Diagram Method

- **Base case:** For all successful $e$

  \[
  e \xrightarrow{\text{program transformation}} e' \xrightarrow{\text{successful}} e'' \xrightarrow{\text{standard reduction steps}} \ldots \xrightarrow{\text{successful}}
  \]

- **General case:** For all programs $e$

  \[
  e \xrightarrow{\text{program transformation}} e' \xrightarrow{\text{standard reduction steps}} e'' \xrightarrow{\text{standard reduction steps}} e''' \xrightarrow{\text{program transformation steps}} \ldots \xrightarrow{\text{successful}}
  \]

- **Inductive construction**
Idea of the Diagram Method

- **Base case:** For all successful $e$
  
  **Inductive construction**

- **General case:** For all programs $e$

\[ e \rightarrow e' \rightarrow e'' \rightarrow \cdots \rightarrow e''' \rightarrow \cdots \rightarrow e'''' \rightarrow \text{successful} \]
Idea of the Diagram Method

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

- **General case:** For all programs $e$
  - $e$ \(\rightarrow\) $e'$
  - $e'$ \(\rightarrow\) $e''$
  - $e''$ \(\rightarrow\) $e'''$

- **Inductive construction**
  - $e$ \(\rightarrow\) $e'$
  - $e'$ \(\rightarrow\) $e''$
  - $e''$ \(\rightarrow\) $e'''$
  - $e'''$ \(\rightarrow\) $e_4$

By the induction hypothesis, successful succ.
Idea of the Diagram Method

- **Base case:** For all successful $e$
  
  $e \xrightarrow{\text{program transformation}} e'$
  
  successful

- **General case:** For all programs $e$
  
  $e \xrightarrow{\text{program transformation}} e'$
  
  standard reduction steps
  
  $e'' \xrightarrow{\text{program transformation steps}} e'''$

- **Inductive construction**
  
  $e \xrightarrow{\text{standard reduction steps}} e''$
  
  $e'' \xrightarrow{\text{program transformation steps}} e'''$
  
  $\ldots$

  $e_4 \xrightarrow{\text{standard reduction steps}} e_5$

  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] and [SSS17, SCP]
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 [NSSSSS07, 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] and [SSS17, SCP]

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)
Representation of the Input

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

The syntax of extended call-by-need lambda-calculi typically includes:

- **lambda-calculus**: variables $x$, abstractions $\lambda x.e$, applications $(e e')$
- **data-constructors** $\text{True, False, Nil, Cons } e_1 e_2, \ldots$
- **data-selectors / case-expressions**
- **let- and recursive let expressions**: `letrec x_1 = e_1, \ldots, x_n = e_n \text{ in } e`
Requirements on the Meta-Syntax

The syntax of extended call-by-need lambda-calculi typically includes:

- lambda-calculus: variables $x$, abstractions $\lambda x.e$, applications $(e e')$
- data-constructors $\text{True, False, Nil, Cons } e_1 e_2, \ldots$
- data-selectors / case-expressions
- let- and recursive let expressions: $\text{letrec } x_1 = e_1, \ldots, x_n = e_n \text{ in } e$

Language LRS parametric over $\mathcal{F}$

**Expressions**

$s \in \text{Expr} ::= \text{var } x \mid \text{letrec } env \text{ in } s \mid (f r_1 \ldots r_{ar(f)})$

where $r_i$ is $o_i, s_i$, or $x_i$ specified by $f \in \mathcal{F}$

**H.O.-Expressions**

$o \in \text{HExpr}^n ::= x_1 \ldots x_n.s$

**Environments**

$env \in \text{Env} ::= \emptyset \mid x = s; env$
Requirements on the Meta-Syntax

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

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

Standard-reduction rules and some program transformations:
\[(\text{SR,}\!\!\!\!\text{lbeta})\ R[(\lambda x.e_1)\ e_2] \rightarrow R[\text{letrec}\ x = e_2\ \text{in}\ e_1] \]
\[(\text{SR,}\!\!\!\!\text{llet})\ \text{letrec}\ Env_1\ \text{in}\ \text{letrec}\ Env_2\ \text{in}\ e \rightarrow \text{letrec}\ Env_1, Env_2\ \text{in}\ e \]
\[ \ldots \]
\[(\text{T,}\!\!\!\!\text{cpx})\ T[\text{letrec}\ x = y, Env\ \text{in}\ C[x]] \rightarrow T[\text{letrec}\ x = y, Env\ \text{in}\ C[y]] \]
\[(\text{T,}\!\!\!\!\text{gc,1})\ T[\text{letrec}\ Env, Env'\ \text{in}\ e] \rightarrow T[\text{letrec}\ Env'\ \text{in}\ e], \]
\[ \text{if } \text{LetVars}(Env) \cap \text{FV}(e, Env') = \emptyset \]
\[(\text{T,}\!\!\!\!\text{gc,2})\ T[\text{letrec}\ Env\ \text{in}\ e] \rightarrow T[e] \quad \text{if } \text{LetVars}(Env) \cap \text{FV}(e) = \emptyset \]
Requirements on the Meta-Syntax

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

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

Standard-reduction rules and some program transformations:

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

(SR,llet) \[ \text{letrec } \text{Env}_1 \text{ in letrec } \text{Env}_2 \text{ in } e \rightarrow \text{letrec } \text{Env}_1, \text{Env}_2 \text{ in } e \]

\[ \cdots \]

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

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

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

Meta-syntax must be capable to represent:

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

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

Reduction contexts:
\[ A ::= [\cdot] | (A \; e) \]
\[ R ::= A | \text{letrec} \; E\text{nv} \; \text{in} \; A | \text{letrec} \{x_i=\text{A}_i[x_{i+1}]\}_{i=1}^{n-1}, x_n=\text{A}_n, E\text{nv}, \text{in} \; A[x_1] \]

Standard-reduction rules and some program transformations:
(SR,\llbeta) \[ R[(\lambda x.e_1) \; e_2] \rightarrow R[\text{letrec} \; x = e_2 \; \text{in} \; e_1] \]
(SR,\lllet) \[ \text{letrec} \; E\text{nv}_1 \; \text{in} \; \text{letrec} \; E\text{nv}_2 \; \text{in} \; e \rightarrow \text{letrec} \; E\text{nv}_1, E\text{nv}_2 \; \text{in} \; e \]
\[ \ldots \]
(T,\llcpx) \[ T[\text{letrec} \; x = y, E\text{nv} \; \text{in} \; C[x]] \rightarrow T[\text{letrec} \; x = y, E\text{nv} \; \text{in} \; C[y]] \]
(T,\llgc,1) \[ T[\text{letrec} \; E\text{nv}, E\text{nv}' \; \text{in} \; e] \rightarrow T[\text{letrec} \; E\text{nv}' \; \text{in} \; e], \]
\[ \text{if LetVars}(E\text{nv}) \cap FV(e, E\text{nv}') = \emptyset \]
(T,\llgc,2) \[ T[\text{letrec} \; E\text{nv} \; \text{in} \; e] \rightarrow T[e] \quad \text{if LetVars}(E\text{nv}) \cap FV(e) = \emptyset \]

Meta-syntax must be capable to represent:

- contexts of different classes
- environments \( E\text{nv}_i \) and environment chains \( \{x_i=\text{A}_i[x_{i+1}]\}_{i=1}^{n-1} \)
Requirements on the Meta-Syntax

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

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

Standard-reduction rules and some program transformations:

(SR,lbeta) \[ R[(\lambda x.e_1)\ e_2] \rightarrow R[\text{letrec } x = e_2 \text{ in } e_1] \]
(SR,llet) \[ \text{letrec } Env_1 \text{ in letrec } Env_2 \text{ in } e \rightarrow \text{letrec } Env_1, Env_2 \text{ in } e \]

(T,cpx) \[ T[\text{letrec } x = y, Env \text{ in } C[x]] \rightarrow T[\text{letrec } x = y, Env \text{ in } C[y]] \]
(T,gc,1) \[ T[\text{letrec } Env, Env' \text{ in } e] \rightarrow T[\text{letrec } Env' \text{ in } e], \]
if \( \text{LetVars}(Env) \cap FV(e, Env') = \emptyset \)
(T,gc,2) \[ T[\text{letrec } Env \text{ in } e] \rightarrow T[e] \] if \( \text{LetVars}(Env) \cap FV(e) = \emptyset \)

Meta-syntax must be capable to represent:

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

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

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

Standard-reduction rules and some program transformations:

1. (SR,\!\!lbeta) \quad R[(\lambda x. e_1) \ e_2] \to R[\text{letrec } x = e_2 \ 	ext{in } e_1]
2. (SR,\!\!llet) \quad \text{letrec } Env_1 \ 	ext{in } \text{letrec } Env_2 \ 	ext{in } e \to \text{letrec } Env_1, \ Env_2 \ 	ext{in } e
3. \ldots
4. (T,cpx) \quad T[\text{letrec } x = y, \ Env \ 	ext{in } C[x]] \to T[\text{letrec } x = y, \ Env \ 	ext{in } C[y]]
5. (T,gc,1) \quad T[\text{letrec } Env, \ Env' \ 	ext{in } e] \to T[\text{letrec } Env' \ 	ext{in } e], \quad \text{if } \text{LetVars}(Env) \cap \text{FV}(e, \ Env') = \emptyset
6. (T,gc,2) \quad T[\text{letrec } Env \ 	ext{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 \( x \in \text{Var} ::= X \) (variable meta-variable)\\ |
\( x \) (concrete variable)

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

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

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

\( Ch[x, s] \) 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

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

\[
\begin{align*}
(T, \text{cpx}) & \quad T[\text{letrec } x = y, \text{Env in } C[x]] \rightarrow T[\text{letrec } x = y, \text{Env in } C[y]] \\
(T, \text{gc}, 1) & \quad T[\text{letrec } \text{Env, Env' in } e] \rightarrow T[\text{letrec } \text{Env' in } e], \\
 & \quad \text{if } \text{LetVars}(\text{Env}) \cap \text{FV}(e, \text{Env'}) = \emptyset \\
(T, \text{gc}, 2) & \quad T[\text{letrec } \text{Env in } e] \rightarrow T[e] \quad \text{if } \text{LetVars}(\text{Env}) \cap \text{FV}(e) = \emptyset
\end{align*}
\]

Restrictions on scoping and emptiness, e.g.:

- (gc): \( \text{Env} \) must not be empty; side condition on variables
- (cpx): \( x, y \) are not captured by \( C \) in \( C[x], C[y] \)
Binding and Scoping Constraints

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

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

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

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

Restrictions on scoping and emptiness, e.g.:

- (gc): \text{Env} must not be empty; side condition on variables
- (cpx): \text{x, y} are not captured by \text{C} in \text{C[x], C[y]}
Binding and Scoping Constraints

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

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

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

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

Restrictions on scoping and emptiness, e.g.:

- (gc): \(\text{Env}\) must not be empty; side condition on variables
- (cpx): \(x, y\) are not captured by \(C\) in \(C[x], C[y]\)
Binding and Scoping Constraints

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

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

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

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

Restrictions on scoping and emptiness, e.g.:

- (gc): Env must not be empty; side condition on variables
- (cpx): x, y are not captured by C in C[x], C[y]
Constraints

A constraint tuple $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ consists of

- non-empty context constraints $\Delta_1$: set of context variables
- non-empty environment constraints $\Delta_2$: set of environment variables
- non-capture constraints (NCCs) $\Delta_3$: set of pairs $(s, d)$
  ($s$ an expression, $d$ a context)

Ground substitution $\rho$ satisfies $(\Delta_1, \Delta_2, \Delta_3)$ iff

- $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$
- $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$
- hole of $\rho(d)$ does not capture variables of $\rho(s)$, for all $(s, d) \in \Delta_3$
Constraints

A constraint tuple $\Delta = (\Delta_1, \Delta_2, \Delta_3)$ consists of
- non-empty context constraints $\Delta_1$: set of context variables
- non-empty environment constraints $\Delta_2$: set of environment variables
- non-capture constraints (NCCs) $\Delta_3$: set of pairs $(s, d)$
  ($s$ an expression, $d$ a context)

Ground substitution $\rho$ satisfies $(\Delta_1, \Delta_2, \Delta_3)$ iff
- $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$
- $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$
- hole of $\rho(d)$ does not capture variables of $\rho(s)$, for all $(s, d) \in \Delta_3$

Example:

$s = \text{letrec } E_1 \text{ in letrec } E_2 \text{ in } S$
$\Delta = (\emptyset, \{E_1, E_2\}, \{(\text{letrec } E_2 \text{ in } S, \text{letrec } E_1 \text{ in } [\cdot])\})$
$\text{semantics}(s, \Delta) =$ nested letrec-expressions with unused outer environment
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

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

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

unifier \( \sigma \) for \( \ell_A \equiv \ell_B \) w.r.t. \( \Delta_A, \Delta_B \)

- As usual, we assume that the meta-variables in \( \ell_A \rightarrow \Delta_A r_A \) are pairwise disjoint from meta-variables in \( \ell_B \rightarrow \Delta_B r_B \) are pairwise disjoint (use fresh copies of the rules)
- Unification also has to treat / respect the constraints \( \Delta := \Delta_A \cup \Delta_B \)
Letrec Unification Problem

A letrec unification problem is a tuple \( P = (\Gamma, \Delta) \) with

- \( \Gamma \): **unification equations** \( s \doteq s' \) of LRSX-expressions
- \( \Delta = (\Delta_1, \Delta_2, \Delta_3) \) is a constraint tuple.

**Occurrence restrictions:**

- Each \( S \)-variable occurs **at most twice** in \( \Gamma \)
- Each \( E \)-, \( Ch \)-, \( D \)-variable occurs **at most once** in \( \Gamma \)
- \( Ch \)-variables are only allowed in one letrec-environment in \( \Gamma \)
Solutions and Unifiers

**Unifier and Solution of** $P = (\Gamma, \Delta)$

A substitution $\rho$ is a **unifier of** $P$ iff

- $\rho(s) \sim_{let} \rho(s')$ for all $s \equiv s' \in \Gamma$
- $\rho$ can be instantiated to satisfy $\Delta$

A unifier $\rho$ is a **solution of** $P$ if $\rho$ is a ground substitution.

$\sim_{let} = $ syntactic equality upto permuting bindings in environments
NP-Hardness

**Theorem (NP-Hardness)**

The decision problem whether a solution for a letrec unification problem exists is NP-hard.

Proof by a reduction from **Monotone one-in-three-3-SAT**.

Sketch: For each clause $C_i = \{S_{i,1}, S_{i,2}, S_{i,3}\}$, add the unification equation

\[
\begin{align*}
\text{letrec } Y_{i,1} &= S_{i,1}; \\
Y_{i,2} &= S_{i,2}; \\
Y_{i,3} &= S_{i,3} \text{ in } c \\
\end{align*}
\]

\[
\begin{align*}
\triangleq \text{letrec } y_{i,1} = \text{false}; \\
y_{i,2} = \text{false}; \\
y_{i,3} = \text{true} \text{ in } c \\
\end{align*}
\]
NP-Hardness

**Theorem (NP-Hardness)**

The decision problem whether a solution for a letrec unification problem exists is NP-hard.

Proof by a reduction from Monotone one-in-three-3-SAT.

Sketch: For each clause $C_i = \{S_{i,1}, S_{i,2}, S_{i,3}\}$, add the unification equation

$$\text{letrec } Y_{i,1} = S_{i,1}; \ Y_{i,2} = S_{i,2}; \ Y_{i,3} = S_{i,3} \text{ in } c$$

$$\equiv \text{letrec } y_{i,1} = false; \ y_{i,2} = false; \ y_{i,3} = true \text{ in } c$$

Remark: Equations have no meta-variables on the right hand side

Matching is already NP-hard.
Unification Algorithm UnifLRS [SSS16, PPDP]

Intermediate **data structure** of the algorithm: \((Sol, \Gamma, \Delta)\) where

- \(Sol\) is a computed substitution
- \(\Gamma\) is a set of equations
- \(\Delta = (\Delta_1, \Delta_2, \Delta_3, \Delta_4)\)
- \((\Delta_1, \Delta_2, \Delta_3)\) are constraints as in a letrec unification problem
- \(\Delta_4\) are environment equations \(E_1; \ldots; E_n = Ch[x, s]\)

**Input:**
For \(P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)\), UnifLRS starts with \((Id, \Gamma, (\Delta_1, \Delta_2, \Delta_3, \emptyset))\)

**Output** (on each branch):
- *Fail* or final state \((Sol, \emptyset, \Delta)\)
Selection of Rules (1)

\[
\frac{(Sol, \Gamma \cup \{x \doteq x\}, \Delta)}{(Sol, \Gamma, \Delta)}
\]

\[
\frac{(Sol, \Gamma \cup \{S \doteq s\}, \Delta)}{(Sol \circ \{S \mapsto s\}, \Gamma[s/S], \Delta[s/S])}
\text{if } S \text{ is not a proper sub-expression of } s
\]

\[
\frac{(Sol, \Gamma \cup \{\text{letrec } env_1 \text{ in } s_1 \doteq \text{letrec } env_2 \text{ in } s_2\}, \Delta)}{(Sol, \Gamma \cup \{env_1 \doteq env_2, s_1 \doteq s_2\}, \Delta)}
\]
Selection of Rules (2)

Unifying bindings and chains:

\[(\text{Sol}, \Gamma \cup \{x = t; \ env_1 \models Ch[y, s]; \ env_2\}, \Delta)\]

\[(\text{Sol} \circ \sigma, \Gamma \cup \{x = t \models y = D[s], \ env_1 \models env_2\}, \Delta \sigma)\]

\[\sigma = \{ Ch[y, s] \mapsto y = D[s] \}\]

“equal”

| (\text{Sol} \circ \sigma, \Gamma \cup \{x = t \models y = D[\text{var} \ Y], \ env_1 \models Ch_2[Y, s]; \ env_2\}, \Delta \sigma) |
| \sigma = \{ Ch_1[y, s] \mapsto y = D[\text{var} \ Y]; \ Ch_2[Y, s] \} |

“prefix”

| (\text{Sol} \circ \sigma, \Gamma \cup \{x = t \models Y_1 = D[\text{var} \ Y_2], \ env_1 \models Ch_1[y, \text{var} \ Y_1]; \ Ch_2[Y_2, s]; \ env_2\}, \Delta \sigma) |
| \sigma = \{ Ch[y, s] \mapsto Ch_1[y, (\text{var} \ Y_1)]; \ Y_1 = D[\text{var} \ Y_2]; \ Ch_2[Y_2, s] \} |

“infix”

| (\text{Sol} \circ \sigma, \Gamma \cup \{x = t \models Y_1 = D[s], \ env_1 \models Ch_2[y, \text{var} \ Y_1]; \ env_2, \Delta \sigma\}) |
| \sigma = \{ Ch_1[y, s] \mapsto Ch_2[y, \text{var} \ Y_1]; \ Y_1 = D[s] \} |

“suffix”
Selection of Failure Rules

**Standard cases:**

\[(Sol, \Gamma \cup \{(x_1 \doteq x_2)\}, \Delta)\]

*Fail*

\[(Sol, \Gamma \cup \{(S \doteq s)\}, \Delta)\]

*Fail* if $S$ is a proper subterm of $s$

**Checking non-capture contraints:**

\[(Sol, \Gamma, (\Delta_1, \Delta_2, \Delta_3 \cup \{(s, d)\}, \Delta_4))\]

*Fail* if $\text{Var}_M(s) \cap \text{CV}_M(d) \neq \emptyset$

\(\text{Var}_M\) and \(\text{CV}_M\) consist of concrete and meta-variables.
Properties of UnifLRS

Proposition (Soundness)
For input $P$ and successful output $(Sol, \emptyset, \Delta)$:
- All ground instances of $Sol$ that do not violate $\Delta$ are solutions of $P$.
- There exists at least one ground instance of $Sol$ which solves $P$.

Proposition (Completeness)
For any solution $\rho$ of a letrec unification problem $P$ there exists a final state $(Sol, \emptyset, \Delta)$ of UnifLRS s.t. $\rho$ is an instance of $Sol$.

Theorem
UnifLRS is sound and complete and terminates in nondeterministic polynomial time and solutions are of polynomial size.
The letrec unification problem is NP-complete.
Computing Joins

Structure of the LRSX-Tool
Computing Diagrams

- $t_1, t_2$ are **meta-expressions** restricted by **constraints** $\nabla$
- computing joins $\to^*$ requires **abstract rewriting** by rules $\ell \to^\Delta r$
- 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 $\Delta$

$$ (t, \nabla) \to (\sigma(r), \nabla \cup \sigma(\Delta)) \quad \text{if} \quad \ell \to^\Delta r, \ t = \sigma(l), \ \text{and} \ \nabla \implies \sigma(\Delta) $$

$\sigma$ is a **matcher** for the letrec matching problem $(\{\ell \subseteq t\}, \Delta, \nabla)$
Matching Algorithm MatchLRS

- For most cases: similar rules as the unification algorithm
- New rules for matching chain-variables, for example matching equations like:
  - \( Ch[x, e]; \text{env} \preceq Ch'[x', e']; \text{env}' \)
  - \( Ch[x, e]; \text{env} \preceq Ch'[x', e']; \text{env}' \)
- New rules for checking that needed constraints \( \Delta_3 \) are implied by given constraints \( \nabla_3 \).

Also infers constraints from the let variable condition:

Example: \texttt{letrec } \( X_1 = S_1; X_2 = S_2 \text{ in } \ldots \) implies validity of the non-capture constraint \((\text{var } X_1, \lambda X_2.[]))\)

**Theorem [Sab17, Unif]**

MatchLRS is sound and complete. The letrec matching problem is NP-complete.
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

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

\[
\begin{align*}
SR, ll & \downarrow \quad \longrightarrow \quad SR, ll \\
\downarrow & \quad \longrightarrow \\
T, gc & \quad \longrightarrow \\
\end{align*}
\]

and the answer diagram

\[
\begin{align*}
Ans & \xrightarrow{T, gc} \quad Ans \\
\end{align*}
\]
Problematic Example: Overlap (SR, llet) and (T, llet)

\[(T, \text{llet}) \quad T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]\]

where an NCC must hold s.t. \(\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset\)

letrec \(E_1\) in
   letrec \(E_2\) in
      letrec \(E_3\) in \(S\)

SR, llet

letrec \(E_1; E_2\) in
   letrec \(E_3\) in \(S\)

Given constraints:
- \(\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset\)
Problematic Example: Overlap (SR,llet) and (T,llet)

\[(T,llet) \to T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \to T[\text{letrec } E; E' \text{ in } S]\]

where an NCC must hold s.t. \(\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset\)

\[
\begin{align*}
\text{letrec } E_1 \text{ in}
\phantom{\text{letrec } E_2 \text{ in}}_1
\phantom{\text{letrec } E_3 \text{ in } S}_2
\text{letrec } E_2 \text{ in}
\phantom{\text{letrec } E_3 \text{ in } S}_3
\text{letrec } E_3 \text{ in } S
\end{align*}
\]

\[
\begin{align*}
\text{SR,llet}
\end{align*}
\]

\[
\begin{align*}
\text{letrec } E_1; E_2 \text{ in}
\phantom{\text{letrec } E_3 \text{ in } S}_1
\phantom{\text{letrec } E_3 \text{ in } S}_2
\text{letrec } E_3 \text{ in } S
\end{align*}
\]

Given constraints:
- \(\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset\)
- \(\text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset\)
Problematic Example: Overlap (SR,llet) and (T,llet)

\((T,llet)\) \(T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]\)

where an NCC must hold s.t. \(\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset\)

\[
\begin{array}{ccc}
\text{letrec } E_1 \text{ in} & T,llet & \text{letrec } E_1 \text{ in} \\
\text{letrec } E_2 \text{ in} & & \text{letrec } E_1 \text{ in} \\
\text{letrec } E_3 \text{ in } S & & \text{letrec } E_2; E_3 \text{ in } S \\
\downarrow & & \downarrow \\
\text{SR,llet} & & \text{SR,llet} \\
\text{letrec } E_1; E_2 \text{ in} & T,llet & \text{letrec } E_1; E_2; E_3 \text{ in } S \\
\text{letrec } E_3 \text{ in } S & & \text{letrec } E_3 \text{ in } S
\end{array}
\]

Given constraints:
- \(\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset\)
- \(\text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset\)
Problematic Example: Overlap (SR,llet) and (T,llet)

\[(T,llet) \rightarrow (T[\text{letrec } E \text{ in } \text{letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S]\]

where an NCC must hold s.t. \(\text{LetVars}(E') \cap \text{Vars}(E) = \emptyset\)

Given constraints:
- \(\text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset\)
- \(\text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset\)

Needed constraints:
- \(\text{LetVars}(E_2; E_3) \cap \text{Vars}(E_1) = \emptyset\)
Problematic Example: Overlap (SR,llet) and (T,llet)

\[ (T,llet) \Rightarrow T[\text{letrec } E \text{ in letrec } E' \text{ in } S] \rightarrow T[\text{letrec } E; E' \text{ in } S] \]

where an NCC must hold s.t. \( \text{LetVars}(E') \cap \text{Vars}(E) = \emptyset \)

Given constraints:
- \( \text{LetVars}(E_2) \cap \text{Vars}(E_1) = \emptyset \)
- \( \text{LetVars}(E_3) \cap \text{Vars}(E_2) = \emptyset \)

Needed constraints:
- \( \text{LetVars}(E_2; E_3) \cap \text{Vars}(E_1) = \emptyset \)
- \( \text{LetVars}(E_3) \cap \text{Vars}(E_1; E_2) = \emptyset \)
An Instance

**Instance:** \( E_1 \mapsto x=z, \ E_2 \mapsto y=1, \ E_3 \mapsto z=2, \ S \mapsto 3 \)

Given constraints:
- \( \text{LetVars}(y=1) \cap \text{Vars}(x=z) = \emptyset \)
- \( \text{LetVars}(z=2) \cap \text{Vars}(y=1) = \emptyset \)

Needed constraints:
- \( \text{LetVars}(y=1; z=2) \cap \text{Vars}(x=z) = \emptyset \)
- \( \text{LetVars}(z=2) \cap \text{Vars}(x=z; y=1) = \emptyset \)

Illegally capturing \( z \)
An Instance

**Instance:** \( E_1 \mapsto x=z, \ E_2 \mapsto y=1, \ E_3 \mapsto z=2, \ S \mapsto 3 \)

Let rec \( x = z \) in
let rec \( y = 1 \) in
let rec \( z = 2 \) in 3

\[ \xrightarrow{SR,llet} \]

Let rec \( x = z \); \( y = 1 \) in
let rec \( z = 2 \) in 3

\[ \xrightarrow{\alpha} \]

\[ \xrightarrow{\alpha} \]

**solution:** use fresh \( \alpha \)-renamings

**Given constraints:**
- \( \text{LetVars}(y = 1) \cap \text{Vars}(x = z) = \emptyset \)
- \( \text{LetVars}(z = 2) \cap \text{Vars}(y = 1) = \emptyset \)

**Needed constraints:**
- \( \text{LetVars}(y_1 = 1; z_1 = 2) \cap \text{Vars}(x_1 = z) = \emptyset \)
- \( \text{LetVars}(z_2 = 2) \cap \text{Vars}(x_2 = z; y_2 = 1) = \emptyset \)
Extending the Method by $\alpha$-Renaming

- $\alpha$-renaming on the meta-level
- Instances must fulfill the distinct variable convention (DVC):

  **Distinct variable convention DVC**

  A ground LRSX-expression fulfills the DVC iff
  - the bound variables are disjoint from the free variables
  - variables on binders are pairwise disjoint

- How to rename meta-variables $X, S, E, D$?
  ⇒ Requires meta-notations for symbolic $\alpha$-renamings
Syntax of the Extended Meta-Language LRSX$\alpha$

Variables
$$x \in \textbf{Var} ::= \langle rc_1, \ldots, rc_n \rangle \cdot X$$ (variable meta-variable)
$$| \quad \langle rc_1, \ldots, rc_n \rangle \cdot x$$ (concrete variable)

Expressions
$$s \in \textbf{Expr} ::= \langle \alpha_{S,i}, rc_1, \ldots, rc_n \rangle \cdot S$$ (expression meta-variable)
$$| \quad \langle \alpha_{D,i}, rc_1, \ldots, rc_n \rangle \cdot D[s]$$ (context meta-variable)
$$| \quad \ldots$$

Environments
$$env \in \textbf{Env} ::= \langle \alpha_{E,i}, rc_1, \ldots, rc_n \rangle \cdot E; env$$ (environment meta-variable)
$$| \quad \ldots$$

a component $\alpha_{U,i}$ $\alpha$-renames instances of $U$

Atomic renaming components
$$rc \in \textbf{ARC} ::= \alpha_{x,i}$$ (fresh renaming of variable $x$)
$$| \quad LV(\alpha_{E,i})$$ (restriction of $\alpha_{E,i}$ on $LetVars(E)$)
$$| \quad CV(\alpha_{D,i})$$ (restriction of $\alpha_{D,i}$ on $CapVars(D)$)
Examples

- $\lambda X.\text{var } X$ is renamed into $\lambda (\alpha_{X,1}) \cdot X.\text{var } (\alpha_{X,1}) \cdot X$

- $\lambda X. S$ is renamed into $\lambda (\alpha_{X,1}) \cdot X. (\alpha_{S,1}, \alpha_{X,1}) \cdot S$

- $\lambda X. \lambda X.\text{var } X$ is renamed into
  
  $\lambda (\alpha_{X,1}) \cdot X. \lambda (\alpha_{X,2}) \cdot X.\text{var } (\alpha_{X,2}, \alpha_{X,1}) \cdot X$ and simplified to
  
  $\lambda (\alpha_{X,1}) \cdot X. \lambda (\alpha_{X,2}) \cdot X.\text{var } (\alpha_{X,2}) \cdot X$

- $\text{letrec } E \text{ in } S$ is renamed into
  
  $\text{letrec } (\alpha_{E,1}) \cdot E \text{ in } (\alpha_{S}, LV(\alpha_{E,1})) \cdot S$
Symbolic $\alpha$-Renaming

Tasks for symbolic $\alpha$-renaming [Sab17, PPDP]:

- A sound **algorithm to $\alpha$-rename** $s \in \text{LRS}_X$ into $AR(s) \in \text{LRS}_X\alpha$
- A sound **matching algorithm** to solve $(s, \nabla) \trianglerighteq (s', \Delta)$ where $s \in \text{LRS}_X$, $s' \in \text{LRS}_X\alpha$
- A sound **test for extended $\alpha$-equivalence** for constrained $\text{LRS}_X\alpha$-expressions
- **Simplification** of $\alpha$-renamings
- **Refreshing $\alpha$-renamings** after rewriting.
Automated Induction

Structure of the LRSX-Tool
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

\[ \begin{array}{c}
\frac{T, gc}{SR, lbeta} \quad \frac{T, gc}{SR, lbeta} \\
\frac{T, gc}{SR, lbeta} \quad \frac{T, gc}{SR, lbeta} \\
\frac{T, gc}{SR, lbeta} \quad \frac{T, gc}{SR, lbeta}
\end{array} \]

\[ Ans \xrightarrow{T, gc} Ans \]
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

\[
\begin{array}{c}
\text{T,gc} \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\text{SR,lbeta} \\
\text{T,gc} \quad \text{SR,lbeta} \\
\text{Ans} \quad \text{T,gc} \quad \text{Ans}
\end{array}
\]

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

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

\[
\begin{array}{c}
T, gc \\
\downarrow SR, lbeta \\
\rightarrow \rightarrow \\
\end{array}
\]

\[
\begin{array}{c}
SR, lbeta \\
\downarrow SR, lbeta \\
\rightarrow \rightarrow \\
T, gc
\end{array}
\]

\[
\begin{array}{c}
Ans \\
\rightarrow \rightarrow \rightarrow \\
T, gc \\
Ans
\end{array}
\]

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

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

- Termination of the string rewrite system implies successful induction

\[
(T, gc), (SR, a_1), \ldots, (SR, a_n), Answer \rightarrow^* (SR, a'_1), \ldots, (SR, a'_m), Answer
\]

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

Obtained TRS:

\[
\begin{align*}
T_{\text{gc}}(\text{SR}_{l\beta}(x)) & \rightarrow \text{SR}_{l\beta}(T_{\text{gc}}(x)) \\
T_{\text{gc}}(\text{SR}_{cp}(x)) & \rightarrow \text{SR}_{cp}(T_{\text{gc}}(x)) \\
T_{\text{gc}}(\text{SR}_{lll}(x)) & \rightarrow \text{SR}_{lll}(T_{\text{gc}}(x)) \\
T_{\text{gc}}(\text{Answer}) & \rightarrow \text{Answer}
\end{align*}
\]

Innermost termination is shown by AProVE and certified by CeTA
Transitive Closures are Required

Example:

\[
\begin{align*}
A[(\lambda X. S) \ S'] & \xleftarrow{T, gc, 2} A[(\text{letrec } E \ \text{in} \ (\lambda X. S)) \ S'] \\
SR, \text{lbeta} & \\
A[\text{letrec } X = S' \ \text{in} \ S]
\end{align*}
\]
Transitive Closures are Required

Example:

\[ A[(\lambda X. S) \ S'] \]

\[ \xrightarrow{T, gc, 2} \]

\[ A[\text{letrec } E \text{ in } (\lambda X. S)) \ S'] \]

\[ \xleftarrow{SR, lbeta} \]

\[ A[\text{letrec } X = S' \text{ in } S] \]
Transitive Closures are Required

Example:

\[
A[(\lambda X. S') S'] \leftarrow T, gc, 2 \quad A[(letrec \ E \ in \ (\lambda X. S)) \ S']
\]

\[
SR, lbeta
\]

\[
A[letrec \ X = S' \ in \ S] \leftarrow T, gc, 2 \quad letrec \ E \ in \ A[(\lambda X. S') S']
\]

\[
SR, lll, +
\]

\[
letrec \ E \ in \ A[(letrec \ X = S' \ in \ S)]
\]

\[
SR, lbeta
\]
Encoding of Transitive Closures

The diagram

\[
\begin{array}{c}
\text{SR,lb}_{\beta} \\
\downarrow \\
\text{T,gc} \\
\downarrow \\
\text{SR,lb}_{\beta} \\
\downarrow \\
\text{T,gc}
\end{array}
\]

is encoded by:

\[
egin{align*}
T_{gc}(SR_{lb}_{\beta}(x)) & \rightarrow \text{gen}(k,x) \\
\text{gen}(s(k),x) & \rightarrow SR_{lll}(\text{gen}(k,x)) \\
\text{gen}(s(k),x) & \rightarrow SR_{lll}(SR_{lb}_{\beta}(T_{gc}(x)))
\end{align*}
\]

- free variable \( k \) on the right hand side to guess the number of steps
- AProVE & CeTA can handle such TRSs
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>Calculus $L_{\text{need}}$ (11 SR rules, 16 transformations, 2 answers)</td>
<td>(\rightarrow)</td>
<td>2242</td>
<td>5425</td>
</tr>
<tr>
<td></td>
<td>(\leftarrow)</td>
<td>3001</td>
<td>7273</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th># overlaps</th>
<th># joins</th>
<th>computation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculus $L_{\text{need}}^{seq}$ (17 SR rules, 18 transformations, 2 answers)</td>
<td>(\rightarrow)</td>
<td>4898</td>
<td>14729</td>
</tr>
<tr>
<td></td>
<td>(\leftarrow)</td>
<td>6437</td>
<td>18089</td>
</tr>
</tbody>
</table>

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

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

Other applications

- Further calculi, for instance, **process calculi** with structural congruence
- Correctness of **translations** between calculi
- Proving **improvements**

Other meta-languages

- **Nominal techniques** to ease reasoning on $\alpha$-renamings:
  - in progress, e.g.
    - Nominal unification for a meta-language with letrec
      [SSKLV16, LOPSTR]
    - Nominal unification for a meta-language with context variables
      [SSS18, FSCD, to appear]

- ...
Thank you!