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**:

$$\texttt{letrec}\ x_1 = s_1; \ldots; x_n = s_n\ \texttt{in}\ t$$

- 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 $\xrightarrow{sr}$

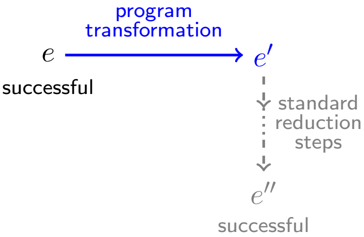
As a core proof method, we need to show

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

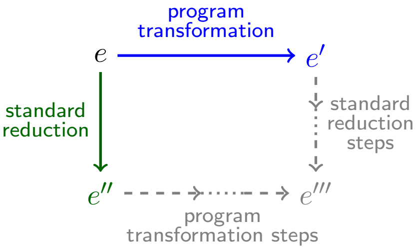
where T' is a contextual closure of T

Idea of the Diagram Method

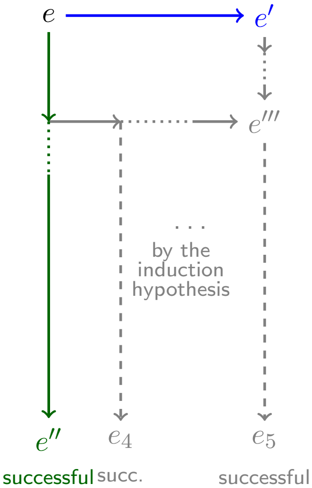
- Base case: For all successful e



- General case: For all programs e



- Inductive construction



Focused Languages and Previous Results

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

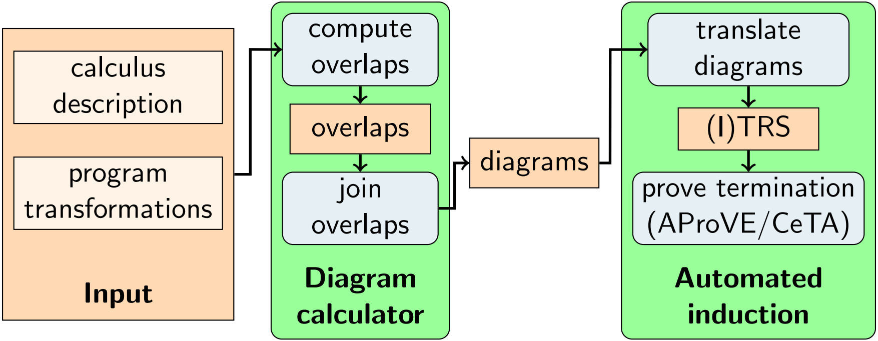
The diagram technique was, for instance, used for

- 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

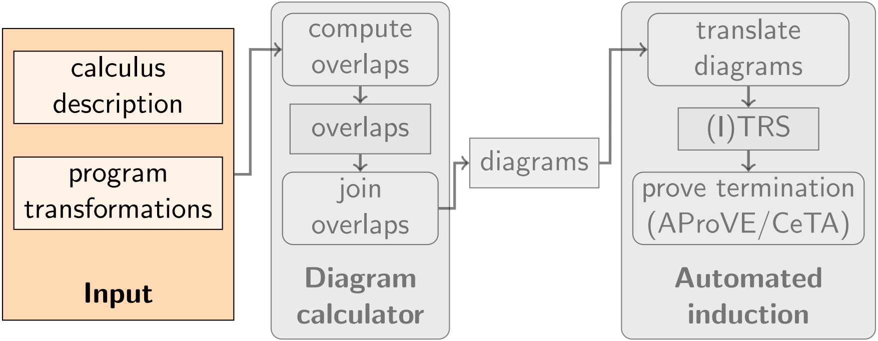
- The diagram method works well
- The method requires to compute overlaps (error-prone, tedious,...)
- Automation of the method would be valuable

Automation of the Diagram-Method



Structure of the LRSX-Tool

Representation of the Input



Structure of the LRSX-Tool

Requirements on the Meta-Syntax

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

Reduction contexts:

$A ::= [\cdot] \mid (A\ e)$

$R ::= A \mid \texttt{letrec}\ Env\ \texttt{in}\ A \mid \texttt{letrec}\ \{x_i{=}A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n{=}A_n, Env, \texttt{in}\ A[x_1]$

Standard-reduction rules and some program transformations:

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

...

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

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

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

Meta-syntax must be capable to represent:

- contexts of different classes
- environments Env_i and environment chains $\{x_i{=}A_i[x_{i+1}]\}_{i=1}^{n-1}$

Syntax of the Meta-Language $\mathtt{LRSX}$

Variables	$x \in \textbf{Var} ::=$	X	(variable meta-variable)
	$\mid$	x	(concrete variable)
Expressions	$s \in \textbf{Expr} ::=$	S	(expression meta-variable)
	$\mid$	$D[s]$	(context meta-variable)
	$\mid$	$\mathtt{letrec}\ env\ \mathtt{in}\ s$	(letrec-expression)
	$\mid$	$\mathtt{var}\ x$	(variable)
	$\mid$	$(f\ r_1 \ldots r_{ar(f)})$	(function application)
		where r_i is $o_i, s_i,$ or x_i specified by f	
	$o \in \textbf{HExpr}^n ::=$	$x_1. \ldots . x_n.s$	(higher-order expression)
Environments	$env \in \textbf{Env} ::=$	$\emptyset$	(empty environment)
	$\mid$	$E; env$	(environment meta-variable)
	$\mid$	$Ch[x, s]; env$	(chain meta-variable)
	$\mid$	$x = s; env$	(binding)

$Ch[x, s]$ represents chains $x{=}C_1[\mathtt{var}\ \mathsf{x}_1];\mathsf{x}_1{=}C_2[\mathtt{var}\ \mathsf{x}_2];\ldots;\mathsf{x}_n{=}C_n[s]$

where C_i are contexts of class $cl(Ch)$

Binding and Scoping Constraints

There are restrictions on scoping and emptiness:

...

(T,cpx) $T[\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[x]] \rightarrow T[\texttt{letrec}\ x = y, Env\ \texttt{in}\ C[y]]$
x, y are not captured by C in $C[x], C[y]$

(T,gc,2) $T[\texttt{letrec}\ Env\ \texttt{in}\ e]\ \rightarrow\ T[e]$ 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 Δ_1: set of context variables
  - ground substitution ρ satisfies $D \in \Delta_1$ iff $\rho(D) \neq [\cdot]$
- non-empty environment constraints Δ_2: set of environment variables
  - ρ satisfies $E \in \Delta_2$ iff $\rho(E) \neq \emptyset$
- non-capture constraints (NCCs) Δ_3: set of pairs (s, d)
  - ρ 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

$$\ell \to_{\Delta} r$$

where ℓ, r are LRSX-expressions and Δ is a constraint-tuple

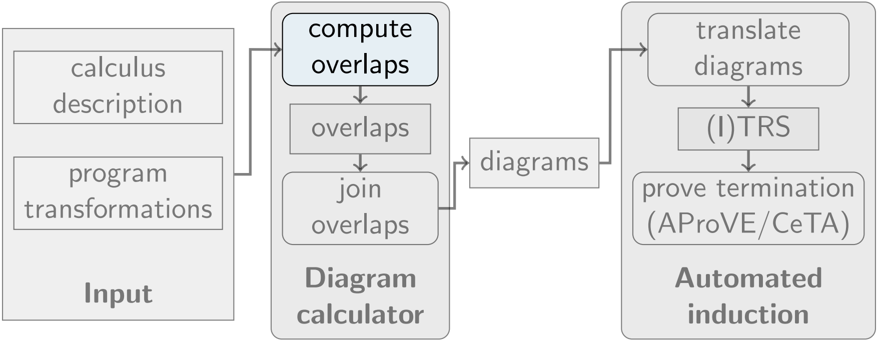
Example:

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

is represented as

$$D[\texttt{letrec}\ E\ \texttt{in}\ S] \to_{(\emptyset,\{E\},\{(S,\texttt{letrec}\ E\ \texttt{in}\ [\cdot])\})} D[S]$$

Computing Overlaps



Structure of the LRSX-Tool

Computing Overlaps by Unification

$$
\begin{tikzcd}
\sigma(\ell_A) = \sigma(\ell_B) \arrow[rr, "\text{program transformation}"] \arrow[d, "\text{standard reduction}"'] & & \sigma(r_B) \arrow[d, dashed, "*"] \\
\sigma(r_A) \arrow[rr, dashed, "*"'] & & \cdot
\end{tikzcd}
$$

unifier σ for $\{\ell_A \doteq \ell_B\}$

Computing Overlaps by Unification

$$
\begin{tikzcd}
\sigma(\ell_A) = \sigma(\ell_B) \arrow[r, "\text{program transformation}"] \arrow[d, "\text{standard reduction}"'] & \sigma(r_B) \arrow[d, dashed, "*"] \\
\sigma(r_A) \arrow[r, dashed, "*"'] & \cdot
\end{tikzcd}
$$

unifier σ for $(\{\ell_A \doteq \ell_B\}, \Delta_A \cup \Delta_B)$

- Unification also has to respect the constraints $\Delta_A \cup \Delta_B$

Computing Overlaps by Unification

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$$
\begin{tikzcd}
\sigma(\ell_A) \arrow[d, "\text{standard reduction}"'] & = & \sigma(\ell_B) \arrow[r, "\text{program transformation}"] & \sigma(r_B) \arrow[d, dashed, "*"] \\
\sigma(r_A) \arrow[rrr, dashed, "*"'] & & & \cdot
\end{tikzcd}
$$

unifier σ 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

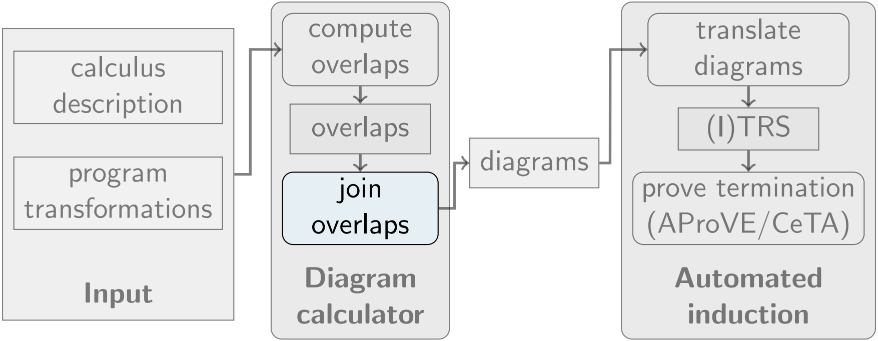
Computing Overlaps by Unification

$$\begin{tikzcd}
\sigma(\ell_A) \; = \; \sigma(\ell_B) \arrow[r, "\text{program transformation}"] \arrow[d, "\text{standard reduction}"'] & (\sigma(r_B), \Delta) \arrow[d, dashed, "*"] \\
(\sigma(r_A), \Delta) \arrow[r, dashed, "*"'] & \cdot
\end{tikzcd}$$

output (σ, Δ) 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

Computing Diagrams

$$\begin{tikzcd}
\cdot \arrow[r, "\text{program transformation}"] \arrow[d, "\text{standard reduction}"'] & (t_1, \nabla) \arrow[d, dashed, "*"] \\
(t_2, \nabla) \arrow[r, dashed, "*"'] & \cdot
\end{tikzcd}$$

- computing joins $\xrightarrow{*}$: **abstract rewriting** by rules $\ell \to_\Delta r$
- meta-variables in ℓ, r are instantiable and meta-variables in t_i are fixed
- rewriting: match ℓ against t_i and show that the given constraints ∇ imply the needed constraints Δ
- Sound and complete matching algorithm MatchLRS [Sab17, UNIF]

Example: (gc)-Transformation

$$\text{(T,gc)} := \text{(T,gc,1)} \cup \text{(T,gc,2)}$$

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

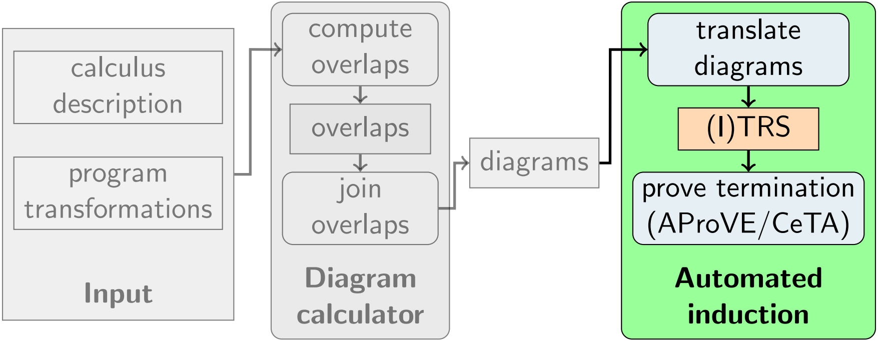
$$\begin{tikzcd}
\cdot \arrow[r, "T{,}gc"] \arrow[d, "SR{,}lbeta"'] & \cdot \arrow[d, dashed, "SR{,}lbeta"] \\
\cdot \arrow[r, dashed, "T{,}gc"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "T{,}gc"] \arrow[d, "SR{,}cp"'] & \cdot \arrow[d, dashed, "SR{,}cp"] \\
\cdot \arrow[r, dashed, "T{,}gc"'] & \cdot
\end{tikzcd}$$

$$\begin{tikzcd}
\cdot \arrow[r, "T{,}gc"] \arrow[d, "SR{,}lll"'] & \cdot \arrow[d, dashed, "SR{,}lll"] \\
\cdot \arrow[r, dashed, "T{,}gc"'] & \cdot
\end{tikzcd}
\qquad
\begin{tikzcd}
\cdot \arrow[r, "T{,}gc"] \arrow[d, "SR{,}lll"'] & \cdot \\
\cdot \arrow[ur, dashed, "T{,}gc"'] &
\end{tikzcd}$$

and the answer diagram

$$Ans \xrightarrow{T,gc} Ans$$

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

$$\begin{tikzcd} \cdot \arrow[r, "{T,gc}"] \arrow[d, "{SR,lbeta}"'] & \cdot \arrow[d, dashed, "{SR,lbeta}"] \\ \cdot \arrow[r, dashed, "{T,gc}"'] & \cdot \end{tikzcd}$$

$$Ans \xrightarrow{T,gc} Ans$$

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

$$\begin{tikzcd} \cdot \arrow[r, "{T,gc}"] \arrow[d, "{SR,lbeta}"'] & \cdot \arrow[d, dashed, "{SR,lbeta}"] \\ \cdot \arrow[r, dashed, "{T,gc}"'] & \cdot \end{tikzcd} \qquad\qquad Ans \xrightarrow{T,gc} Ans$$

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

$$(T, gc), (SR, lbeta) \to (SR, lbeta), (T, gc) \qquad\qquad (T, gc), Answer \to Answer$$

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

$$\begin{tikzcd} \cdot \arrow[r, "T{,}gc"] \arrow[d, "SR{,}lbeta"'] & \cdot \arrow[d, dashed, "SR{,}lbeta"] \\ \cdot \arrow[r, dashed, "T{,}gc"'] & \cdot \end{tikzcd} \qquad\qquad Ans \xrightarrow{T,gc} Ans$$

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

$$(T, gc), (SR, lbeta) \to (SR, lbeta), (T, gc) \qquad\qquad (T, gc), Answer \to Answer$$

- Termination of the string rewrite system implies successful induction
- We use term rewrite systems and innermost-termination and apply AProVE and certifier CeTA

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Symbolic α-Renaming

- Joining overlaps requires α-renaming

$$
\begin{array}{ccc}
(\lambda X.S)\ (\texttt{letrec}\ E_1\ \texttt{in}\ S') & \xleftarrow{T,gc} & (\lambda X.S)\ (\texttt{letrec}\ E_1;E_2\ \texttt{in}\ S') \\
\downarrow{\scriptstyle sr,lbeta} & & \downarrow{\scriptstyle sr,lbeta} \\
\begin{array}{l}\texttt{letrec} \\ \ X{=}(\texttt{letrec}\ E_1\ \texttt{in}\ S') \\ \texttt{in}\ S\end{array} & & \texttt{letrec}\ X{=}(\texttt{letrec}\ E_1;E_2\ \texttt{in}\ S')\ \texttt{in}\ S
\end{array}
$$

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

- Solution: Extend the meta-language and algorithms with **symbolic** α-renamings [Sab17,PPDP]

Advanced Techniques (continued)

Transitive Closures

- Transitive closures of reduction / transformation rules, e.g.
  $A[\texttt{letrec}\ Env\ \texttt{in}\ s] \xrightarrow{sr,+} \texttt{letrec}\ Env\ \texttt{in}\ A[s]$
- 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/LRSXTOOL61`
- computes diagrams and performs the automated induction

	# overlaps	# joins	computation time
Calculus L_{need} (11 SR rules, 16 transformations, 2 answers)			
$\rightarrow$	2242	5425	48 secs.
$\leftarrow$	3001	7273	116 secs.
Calculus L_{need}^{+seq} (17 SR rules, 18 transformations, 2 answers)			
$\rightarrow$	4898	14729	149 secs.
$\leftarrow$	6437	18089	255 secs.
Calculus LR (76 SR rules, 43 transformations, 17 answers)			
$\rightarrow$	87041	391264	$\sim$ 19 hours
$\leftarrow$	107333	429104	$\sim$ 16 hours

Conclusion and Outlook

Conclusion

- Automation of the diagram method for meta-language LRSX
- Algorithms for unification, matching, symbolic α-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 α-renamings:
  - Nominal unification with letrec [SSKLV16, LOPSTR]
  - Nominal unification with context variables [SSS18, FSCD]

Thank you!