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

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- **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

Motivation

A program transformation is a binary relation on program fragments

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

$\xrightarrow{T}$

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

Motivation

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

A program transformation is a binary relation on program fragments

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

$\xrightarrow{T}$

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

Some applications:

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

Correctness

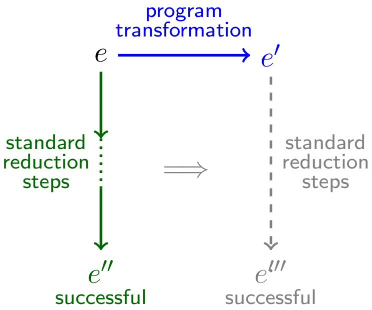
Program transformation T is correct iff $T \subseteq \sim_c$

- 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:

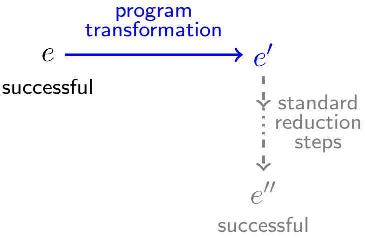


Idea of the Diagram Method

- Base case: For all successful e

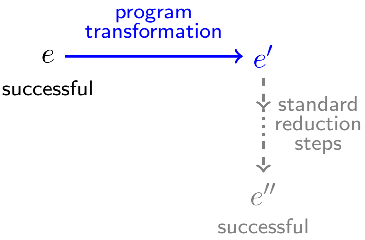
Idea of the Diagram Method

- Base case: For all successful e

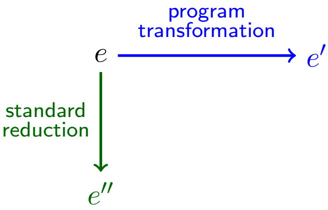


Idea of the Diagram Method

- Base case: For all successful e

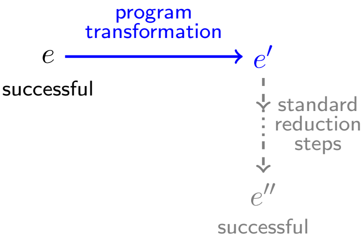


- General case: For all programs e

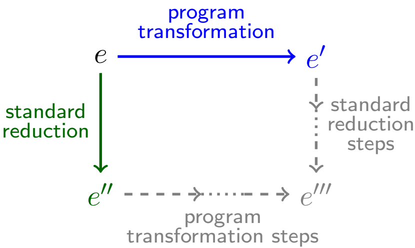


Idea of the Diagram Method

- Base case: For all successful e

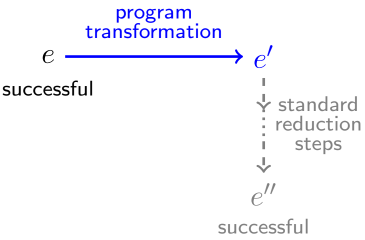


- General case: For all programs e

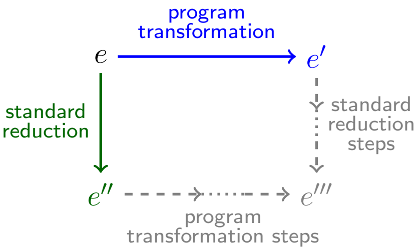


Idea of the Diagram Method

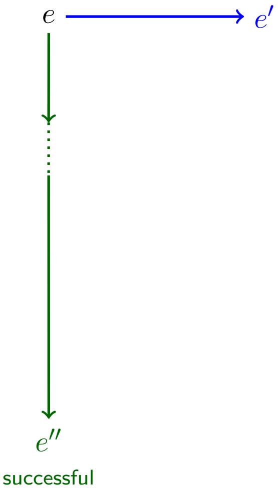
- Base case: For all successful e



- General case: For all programs e

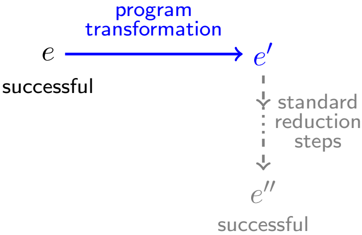


- Inductive construction

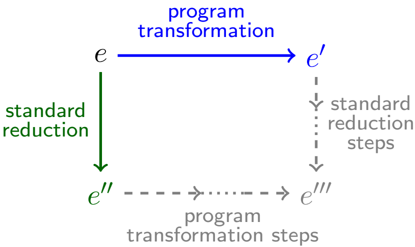


Idea of the Diagram Method

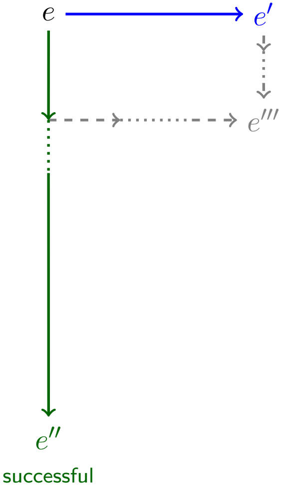
- Base case: For all successful e



- General case: For all programs e

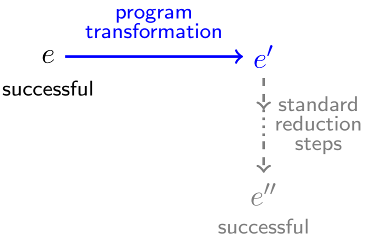


- Inductive construction

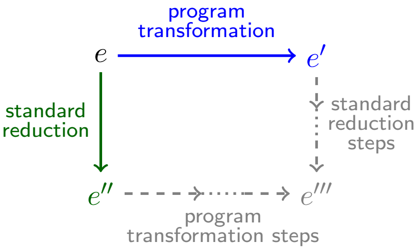


Idea of the Diagram Method

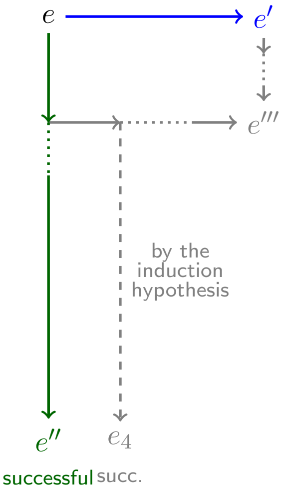
- Base case: For all successful e



- General case: For all programs e

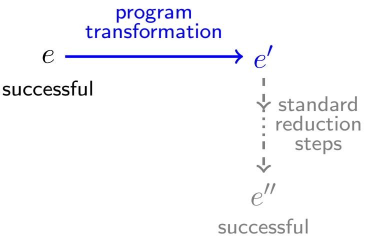


- Inductive construction

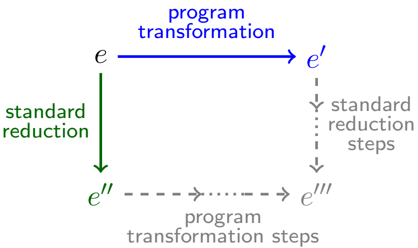


Idea of the Diagram Method

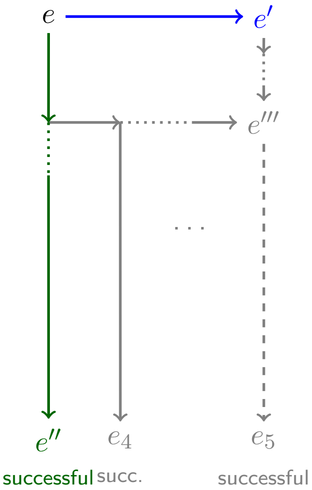
- Base case: For all successful e



- 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] and [SSS17, SCP]

Focused Languages and Previous Results

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

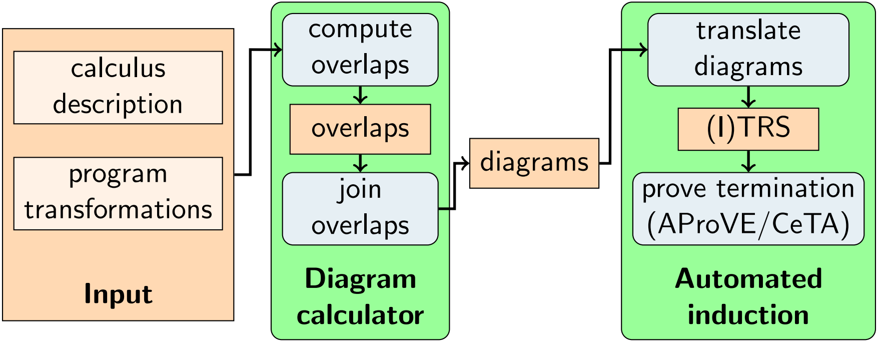
The diagram technique was, for instance, used for

- 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] 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

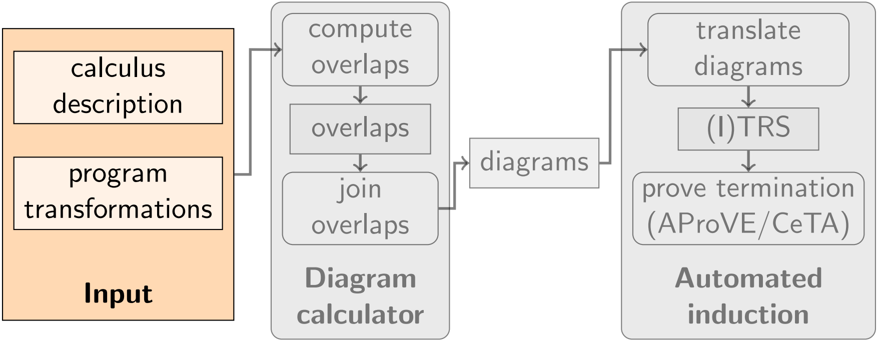
Automation of the Diagram-Method



Structure of the LRSX-Tool

Representation of the Input

GOETHE UNIVERSITÄT FRANKFURT AM MAIN



Structure of the LRSX-Tool

Requirements on the Meta-Syntax

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

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

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 $\mathtt{True}, \mathtt{False}, \mathtt{Nil}, \mathtt{Cons}\ e_1\ e_2, \ldots$
- data-selectors / case-expressions
- let- and recursive let expressions: $\mathtt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \mathtt{in}\ e$

Language LRS parametric over $\mathcal{F}$

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

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

H.O.-Expressions $o \in \textbf{HExpr}^n ::= \mathsf{x}_1 \ldots . \mathsf{x}_n.s$

Environments $env \in \textbf{Env} ::= \emptyset \mid x = s; env$

Requirements on the Meta-Syntax

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) $R[(\lambda x.e_1)\ e_2] \rightarrow R[\texttt{letrec}\ x = e_2\ \texttt{in}\ e_1]$

(SR,llet) $\texttt{letrec}\ Env_1\ \texttt{in}\ \texttt{letrec}\ Env_2\ \texttt{in}\ e \rightarrow \texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e$

...

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

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

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

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

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

...

(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 $\{x_i{=}A_i[x_{i+1}]\}_{i=1}^{n-1}$

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

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

...

(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 $\{x_i{=}A_i[x_{i+1}]\}_{i=1}^{n-1}$

Requirements on the Meta-Syntax

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) $R[(\lambda x.e_1)\ e_2] \rightarrow R[\texttt{letrec}\ x = e_2\ \texttt{in}\ e_1]$

(SR,llet) $\texttt{letrec}\ Env_1\ \texttt{in}\ \texttt{letrec}\ Env_2\ \texttt{in}\ e \rightarrow \texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e$

...

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

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

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

Meta-syntax must be capable to represent:

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

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) $R[(\lambda x.e_1)\ e_2] \rightarrow R[\texttt{letrec}\ x = e_2\ \texttt{in}\ e_1]$

(SR,llet) $\texttt{letrec}\ Env_1\ \texttt{in}\ \texttt{letrec}\ Env_2\ \texttt{in}\ e \rightarrow \texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e$

...

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

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

(T,gc,2) $T[\texttt{letrec}\ Env\ \texttt{in}\ e] \rightarrow T[e]$ 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 LRSX

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Variables	$x \in \textbf{Var} ::=$	X	(variable meta-variable)
	$\mid$	$\mathtt{x}$	(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 $x{=}C_1[\mathtt{var}\ \mathtt{x}_1];\mathtt{x}_1{=}C_2[\mathtt{var}\ \mathtt{x}_2];\ldots;\mathtt{x}_n{=}C_n[s]$

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

Binding and Scoping Constraints

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

...

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

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

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

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 $C[x], C[y]$

Binding and Scoping Constraints

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

...

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

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

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

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 $C[x], C[y]$

Binding and Scoping Constraints

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

...

(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 $LetVars(Env) \cap FV(e) = \emptyset$

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 $C[x], C[y]$

Binding and Scoping Constraints

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

...

(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 $LetVars(Env) \cap FV(e) = \emptyset$

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

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

- non-empty context constraints Δ_1: set of context variables
- non-empty environment constraints Δ_2: set of environment variables
- non-capture constraints (NCCs) Δ_3: set of pairs (s, d)
  (s an expression, d a context)

Ground substitution ρ **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 Δ_1: set of context variables
- non-empty environment constraints Δ_2: set of environment variables
- non-capture constraints (NCCs) Δ_3: set of pairs (s, d)
  (s an expression, d a context)

Ground substitution ρ **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:

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

Representation of Rules

Standard reductions and transformations are represented as

$$\ell \rightarrow_{\Delta} r$$

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

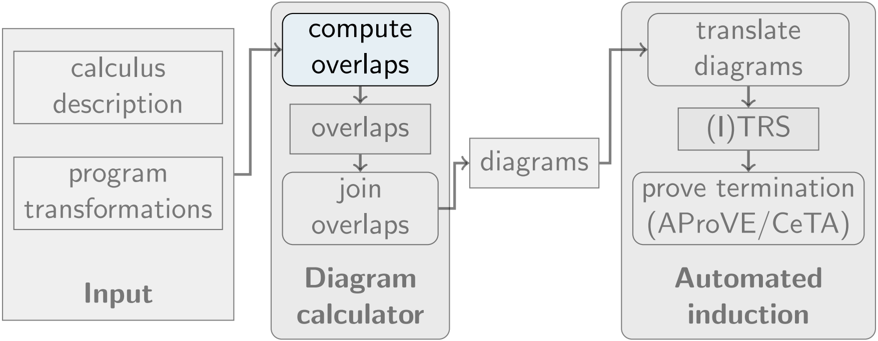
Example:

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

is represented as

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

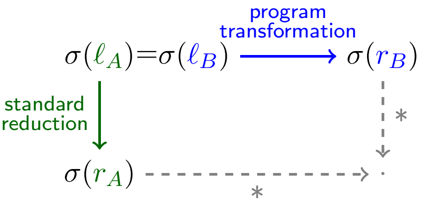
Computing Overlaps



Structure of the LRSX-Tool

Computing Overlaps by Unification

GOETHE UNIVERSITÄT FRANKFURT AM MAIN



unifier σ for $\ell_A \doteq \ell_B$ w.r.t. Δ_A, Δ_B

- As usual, we assume that the meta-variables in $\ell_A \to_{\Delta_A} r_A$ are pairwise disjoint from meta-variables in $\ell_B \to_{\Delta_B} r_B$ 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

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

- Γ: **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 Γ
- Each E-, Ch-, D-variable occurs at most once in Γ
- Ch-variables are only allowed in one letrec-environment in Γ

Solutions and Unifiers

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

A substitution ρ is a **unifier of P** iff

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

A unifier ρ is a **solution of P** if ρ 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

$$\begin{array}{rl} & \texttt{letrec}\ Y_{i,1} = S_{i,1};\ \ Y_{i,2} = S_{i,2};\ \ Y_{i,3} = S_{i,3}\ \texttt{in}\ c \\ \doteq & \texttt{letrec}\ \mathsf{y}_{i,1} = \mathit{false};\ \mathsf{y}_{i,2} = \mathit{false};\ \mathsf{y}_{i,3} = \mathit{true}\ \texttt{in}\ c \end{array}$$

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

$$\begin{array}{rl} & \texttt{letrec}\ Y_{i,1} = S_{i,1};\ \ Y_{i,2} = S_{i,2};\ \ Y_{i,3} = S_{i,3}\ \texttt{in}\ c \\ \doteq & \texttt{letrec}\ \mathsf{y}_{i,1} = \mathit{false};\ \mathsf{y}_{i,2} = \mathit{false};\ \mathsf{y}_{i,3} = \mathit{true}\ \texttt{in}\ c \end{array}$$

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, Γ, Δ) where

- Sol is a computed substitution
- Γ 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
- Δ_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)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$$\frac{(Sol, \Gamma \dot\cup \{\mathsf{x} \doteq \mathsf{x}\}, \Delta)}{(Sol, \Gamma, \Delta)}$$

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

$$\frac{(Sol, \Gamma \dot\cup \{\texttt{letrec}\ env_1\ \texttt{in}\ s_1 \doteq \texttt{letrec}\ env_2\ \texttt{in}\ s_2\}, \Delta)}{(Sol, \Gamma \dot\cup \{env_1 \doteq env_2, s_1 \doteq s_2\}, \Delta)}$$

Selection of Rules (2)

Unifying bindings and chains:

$$\frac{(Sol, \Gamma \dot\cup \{x = t; env_1 \doteq Ch[y,s]; env_2\}, \Delta)}{\begin{array}{ll} (Sol \circ \sigma, \Gamma \dot\cup \{x = t \doteq y = D[s], env_1 \doteq env_2\}, \Delta\sigma) & \\ \sigma = \{Ch[y,s] \mapsto y = D[s]\} & \text{"equal"} \\ \mid (Sol \circ \sigma, \Gamma \dot\cup \{x = t \doteq y = D[\mathtt{var}\ Y], env_1 \doteq Ch_2[Y,s]; env_2\}, \Delta\sigma) & \\ \sigma = \{Ch_1[y,s] \mapsto y = D[\mathtt{var}\ Y]; Ch_2[Y,s]\} & \text{"prefix"} \\ \mid (Sol \circ \sigma, \Gamma \dot\cup \{x = t \doteq Y_1 = D[\mathtt{var}\ Y_2], env_1 \doteq Ch_1[y, \mathtt{var}\ Y_1]; Ch_2[Y_2,s]; env_2\}, \Delta\sigma) & \\ \sigma = \{Ch[y,s] \mapsto Ch_1[y, (\mathtt{var}\ Y_1)]; Y_1 = D[\mathtt{var}\ Y_2]; Ch_2[Y_2,s]\} & \text{"infix"} \\ \mid (Sol \circ \sigma, \Gamma \dot\cup \{x = t \doteq Y_1 = D[s], env_1 \doteq Ch_2[y, \mathtt{var}\ Y_1]; env_2, \Delta\sigma\}) & \\ \sigma = \{Ch_1[y,s] \mapsto Ch_2[y, \mathtt{var}\ Y_1]; Y_1 = D[s]\} & \text{"suffix"} \end{array}}$$

Selection of Failure Rules

Standard cases:

$$\frac{(Sol, \Gamma \dot\cup \{(\mathsf{x}_1 \doteq \mathsf{x}_2)\}, \Delta)}{\textit{Fail}}$$

$$\frac{(Sol, \Gamma \dot\cup \{(S \doteq s)\}, \Delta)}{\textit{Fail}} \quad \text{if } S \text{ is a proper subterm of } s$$

Checking non-capture contraints:

$$\frac{(Sol, \Gamma, (\Delta_1, \Delta_2, \Delta_3 \cup \{(s, d)\}, \Delta_4))}{\textit{Fail}} \quad \text{if } Var_M(s) \cap CV_M(d) \neq \emptyset$$

Var_M and 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 Δ are solutions of P.
- There exists at least one ground instance of Sol which solves P.

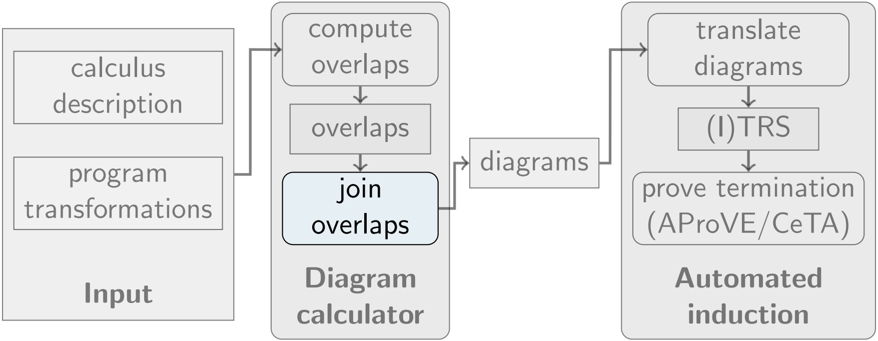
Proposition (Completeness)

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

Theorem

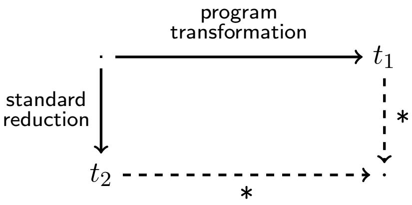
UnifLRS is sound and complete and terminates in nondeterministic polynomial time and solutions are of polynomial size.
The letrec unification problem is NP-complete.

Computing Joins



Structure of the LRSX-Tool

Computing Diagrams



- t_1, t_2 are **meta-expressions** restricted by **constraints** ∇
- computing joins $\xrightarrow{*}$ requires **abstract rewriting** by rules $\ell \rightarrow_{\Delta} r$
- 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

$$(t, \nabla) \rightarrow (\sigma(r), \nabla \cup \sigma(\Delta)) \qquad \text{if } \ell \rightarrow_{\Delta} r,\ t = \sigma(l), \text{ and } \nabla \implies \sigma(\Delta)$$

σ is a **matcher** for the **letrec matching problem** $(\{\ell \trianglelefteq t\}, \Delta, \nabla)$

Matching Algorithm MatchLRS

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- For most cases: similar rules as the unification algorithm
- New rules for matching chain-variables, for example matching equations like:
  - $Ch[x,e]; env \trianglelefteq Ch'[x',e']; env'$
  - $Ch[x,e]; env \trianglelefteq Ch'[x',e']; env'$
- New rules for checking that needed constraints Δ_3 are implied by given constraints ∇_3.
  Also infers constraints from the let variable condition:
  Example: `letrec` $X_1 = S_1; X_2 = S_2$ `in` $\ldots$ implies validity of the non-capture constraint $(\texttt{var}\ X_1, \lambda X_2.[])$

Theorem [Sab17, Unif]

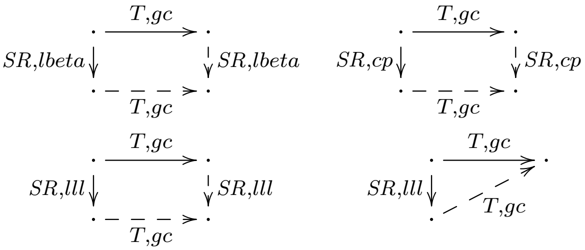
MatchLRS is sound and complete. The letrec matching problem is NP-complete.

Example: (gc)-Transformation

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

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



and the answer diagram

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

Problematic Example: Overlap (SR,llet) and (T,llet)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

(T,llet) $T[\texttt{letrec}\ E\ \texttt{in}\ \texttt{letrec}\ E'\ \texttt{in}\ S] \rightarrow T[\texttt{letrec}\ E; E'\ \texttt{in}\ S]$
where an NCC must hold s.t. $LetVars(E') \cap Vars(E) = \emptyset$

$\texttt{letrec}\ E_1\ \texttt{in}$
$\quad \texttt{letrec}\ E_2\ \texttt{in}$
$\quad\quad \texttt{letrec}\ E_3\ \texttt{in}\ S$

$\downarrow$ SR,llet

$\texttt{letrec}\ E_1; E_2\ \texttt{in}$
$\quad \texttt{letrec}\ E_3\ \texttt{in}\ S$

Given constraints:
- $LetVars(E_2) \cap Vars(E_1) = \emptyset$

Problematic Example: Overlap (SR,llet) and (T,llet)

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

(T,llet) $T[\texttt{letrec}\ E\ \texttt{in}\ \texttt{letrec}\ E'\ \texttt{in}\ S] \rightarrow T[\texttt{letrec}\ E; E'\ \texttt{in}\ S]$
where an NCC must hold s.t. $LetVars(E') \cap Vars(E) = \emptyset$

$\texttt{letrec}\ E_1\ \texttt{in}$
$\quad\texttt{letrec}\ E_2\ \texttt{in}$
$\quad\quad\texttt{letrec}\ E_3\ \texttt{in}\ S$

$\xrightarrow{\text{T,llet}}$

$\texttt{letrec}\ E_1\ \texttt{in}$
$\quad\texttt{letrec}\ E_2; E_3\ \texttt{in}\ S$

SR,llet $\downarrow$

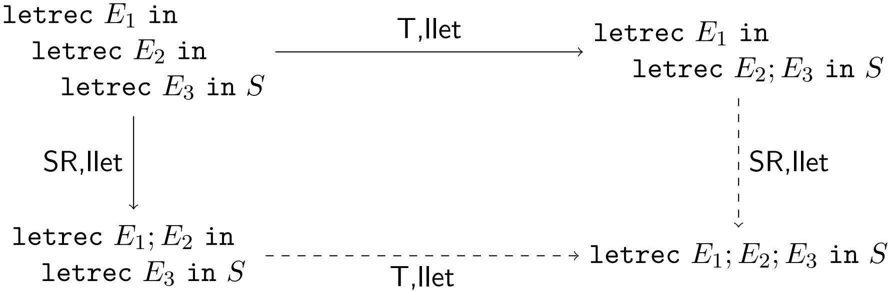
$\texttt{letrec}\ E_1; E_2\ \texttt{in}$
$\quad\texttt{letrec}\ E_3\ \texttt{in}\ S$

Given constraints:

- $LetVars(E_2) \cap Vars(E_1) = \emptyset$
- $LetVars(E_3) \cap Vars(E_2) = \emptyset$

Problematic Example: Overlap (SR,llet) and (T,llet)

(T,llet) $T[\texttt{letrec}\ E\ \texttt{in}\ \texttt{letrec}\ E'\ \texttt{in}\ S] \to T[\texttt{letrec}\ E; E'\ \texttt{in}\ S]$
where an NCC must hold s.t. $LetVars(E') \cap Vars(E) = \emptyset$

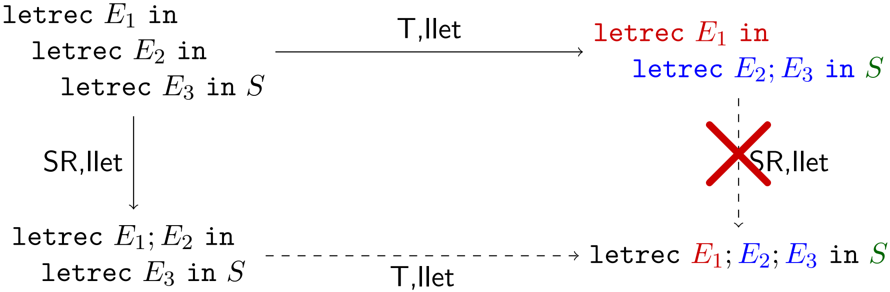


Given constraints:

- $LetVars(E_2) \cap Vars(E_1) = \emptyset$
- $LetVars(E_3) \cap Vars(E_2) = \emptyset$

Problematic Example: Overlap (SR,llet) and (T,llet)

(T,llet) $T[\texttt{letrec}\ E\ \texttt{in}\ \texttt{letrec}\ E'\ \texttt{in}\ S] \rightarrow T[\texttt{letrec}\ E; E'\ \texttt{in}\ S]$
where an NCC must hold s.t. $LetVars(E') \cap Vars(E) = \emptyset$



Given constraints:

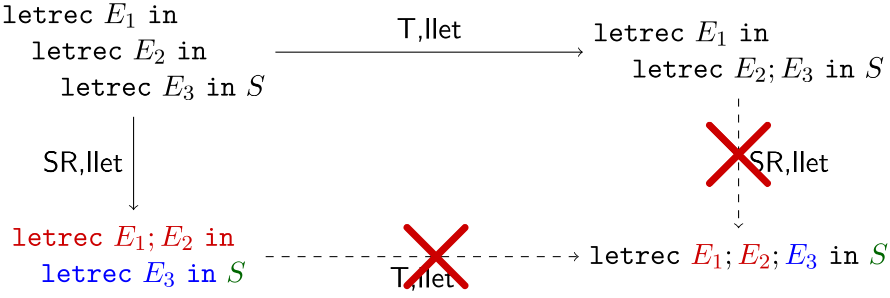
- $LetVars(E_2) \cap Vars(E_1) = \emptyset$
- $LetVars(E_3) \cap Vars(E_2) = \emptyset$

Needed constraints:

- $LetVars(E_2; E_3) \cap Vars(E_1) = \emptyset$

Problematic Example: Overlap (SR,llet) and (T,llet)

(T,llet) $T[\texttt{letrec}\ E\ \texttt{in}\ \texttt{letrec}\ E'\ \texttt{in}\ S] \rightarrow T[\texttt{letrec}\ E; E'\ \texttt{in}\ S]$
where an NCC must hold s.t. $LetVars(E') \cap Vars(E) = \emptyset$



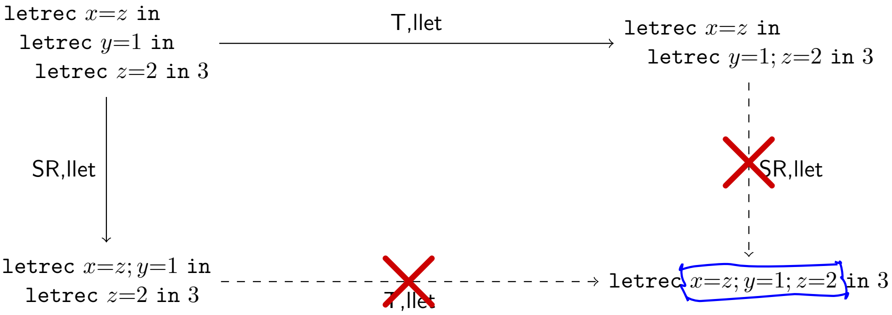
Given constraints:
- $LetVars(E_2) \cap Vars(E_1) = \emptyset$
- $LetVars(E_3) \cap Vars(E_2) = \emptyset$

Needed constraints:
- $LetVars(E_2; E_3) \cap Vars(E_1) = \emptyset$
- $LetVars(E_3) \cap Vars(E_1; E_2) = \emptyset$

$\not\Longrightarrow$

An Instance

Instance: $E_1 \mapsto x{=}z, \quad E_2 \mapsto y{=}1, \quad E_3 \mapsto z{=}2, \quad S \mapsto 3$



illegal capture of z

Given constraints:

- $\mathit{LetVars}(y{=}1) \cap \mathit{Vars}(x{=}z) = \emptyset$
- $\mathit{LetVars}(z{=}2) \cap \mathit{Vars}(y{=}1) = \emptyset$

Needed constraints:

- $\mathit{LetVars}(y{=}1; z{=}2) \cap \mathit{Vars}(x{=}z) = \emptyset$
- $\mathit{LetVars}(z{=}2) \cap \mathit{Vars}(x{=}z; y{=}1) = \emptyset$

An Instance

Instance: $E_1 \mapsto x{=}z, \quad E_2 \mapsto y{=}1, \quad E_3 \mapsto z{=}2, \quad S \mapsto 3$

$\texttt{letrec}\ x{=}z\ \texttt{in}\ \texttt{letrec}\ y{=}1\ \texttt{in}\ \texttt{letrec}\ z{=}2\ \texttt{in}\ 3 \xrightarrow{\text{T,llet}} \texttt{letrec}\ x{=}z\ \texttt{in}\ \texttt{letrec}\ y{=}1; z{=}2\ \texttt{in}\ 3$

$\downarrow \alpha$

$\texttt{letrec}\ x_2{=}z\ \texttt{in}\ \texttt{letrec}\ y_2{=}1; z_2{=}2\ \texttt{in}\ 3$

$\downarrow$ SR,llet

$\texttt{letrec}\ x_2{=}z; y_2{=}1; z_2{=}2\ \texttt{in}\ 3$

$\sim_\alpha$

$\texttt{letrec}\ x_1{=}z; y_1{=}1; z_1{=}2\ \texttt{in}\ 3$

SR,llet (from $\texttt{letrec}\ x{=}z\ \texttt{in}\ \texttt{letrec}\ y{=}1\ \texttt{in}\ \texttt{letrec}\ z{=}2\ \texttt{in}\ 3$):

$\texttt{letrec}\ x{=}z; y{=}1\ \texttt{in}\ \texttt{letrec}\ z{=}2\ \texttt{in}\ 3 \xrightarrow{\alpha} \texttt{letrec}\ x_1{=}z; y_1{=}1\ \texttt{in}\ \texttt{letrec}\ z_1{=}2\ \texttt{in}\ 3 \xrightarrow{\text{T,llet}} \texttt{letrec}\ x_1{=}z; y_1{=}1; z_1{=}2\ \texttt{in}\ 3$

solution: use fresh α-renamings

Given constraints:

- $\mathit{LetVars}(y{=}1) \cap \mathit{Vars}(x{=}z) = \emptyset$
- $\mathit{LetVars}(z{=}2) \cap \mathit{Vars}(y{=}1) = \emptyset$

Needed constraints:

- $\mathit{LetVars}(y_1{=}1; z_1{=}2) \cap Vars(x_1{=}z) = \emptyset$
- $\mathit{LetVars}(z_2{=}2) \cap \mathit{Vars}(x_2{=}z; y_2{=}1) = \emptyset$

Extending the Method by α-Renaming

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

- α-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?

$\Rightarrow$ Requires meta-notations for symbolic α-renamings

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

Variables

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

Expressions

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

Environments

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

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

Atomic renaming components

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

Examples

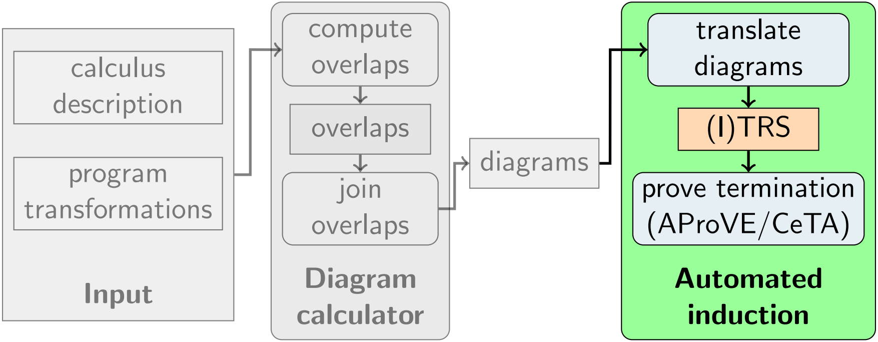
- $\lambda X.\texttt{var}\ X$ is renamed into $\lambda\langle\alpha_{X,1}\rangle\cdot X.\texttt{var}\ \langle\alpha_{X,1}\rangle\cdot X$
- $\lambda X.S$ is renamed into $\lambda\langle\alpha_{X,1}\rangle\cdot X.\langle\alpha_{S,1},\alpha_{X,1}\rangle\cdot S$
- $\lambda X.\lambda X.\texttt{var}\ X$ is renamed into $\lambda\langle\alpha_{X,1}\rangle\cdot X.\lambda\langle\alpha_{X,2}\rangle\cdot X.\texttt{var}\ \langle\alpha_{X,2},\alpha_{X,1}\rangle\cdot X$ and simplified to $\lambda\langle\alpha_{X,1}\rangle\cdot X.\lambda\langle\alpha_{X,2}\rangle\cdot X.\texttt{var}\ \langle\alpha_{X,2}\rangle\cdot X$
- $\texttt{letrec}\ E\ \texttt{in}\ S$ is renamed into $\texttt{letrec}\ \langle\alpha_{E,1}\rangle\cdot E\ \texttt{in}\ \langle\alpha_S, LV(\alpha_{E,1})\rangle\cdot S$

Symbolic α-Renaming

Tasks for symbolic α-renaming [Sab17, PPDP]:

- A sound **algorithm to α-rename** $s \in \texttt{LRSX}$ into $AR(s) \in \texttt{LRSX}\alpha$
- A sound **matching algorithm** to solve $(s, \nabla) \trianglelefteq (s', \Delta)$ where $s \in \texttt{LRSX}$, $s' \in \texttt{LRSX}\alpha$
- A sound **test for extended α-equivalence** for constrained $\texttt{LRSX}\alpha$-expressions
- **Simplification** of α-renamings
- **Refreshing α-renamings** after rewriting.

Automated Induction



Structure of the LRSX-Tool

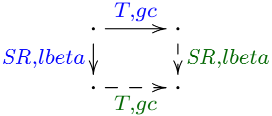
Automated Induction: Ideas [RSSS12, IJCAR]

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

$$\begin{array}{ccc} \cdot & \xrightarrow{T,gc} & \cdot \\ {\scriptstyle SR,lbeta}\downarrow & & \downarrow{\scriptstyle SR,lbeta} \\ \cdot & \underset{T,gc}{\dashrightarrow} & \cdot \end{array} \qquad\qquad Ans \xrightarrow{T,gc} Ans$$

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



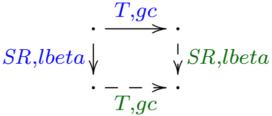
$$Ans \xrightarrow{T,gc} Ans$$

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

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

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



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

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

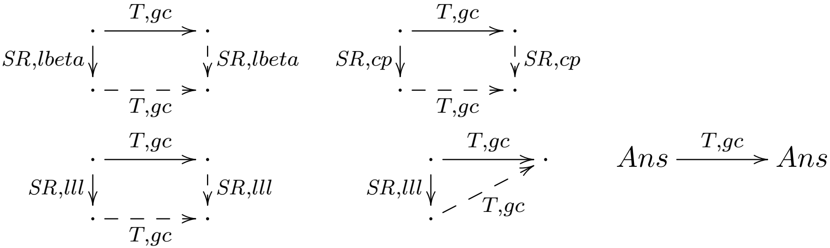
$$(T, gc), (SR, lbeta) \to (SR, lbeta), (T, gc) \qquad (T, gc), Answer \to Answer$$

- Termination of the string rewrite system implies successful induction

$$(T, gc), (SR, a_1), \ldots, (SR, a_n), Answer \xrightarrow{*} (SR, a'_1), \ldots, (SR, a'_m), Answer$$

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

Example



Obtained TRS:

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

Innermost termination is shown by AProVE and certified by CeTA

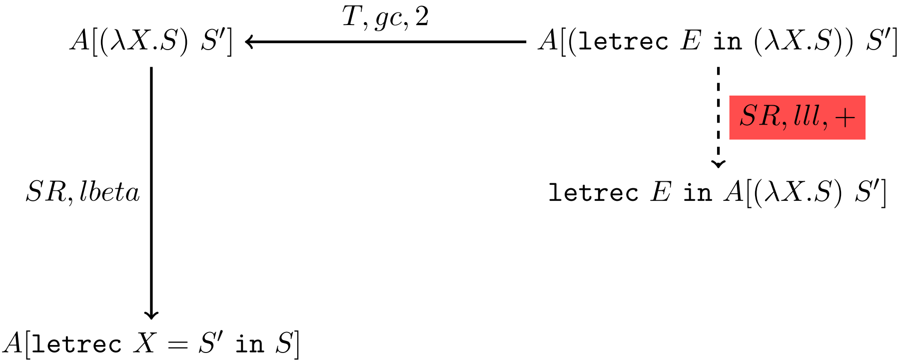
Transitive Closures are Required

Example:

$$A[(\lambda X.S)\ S'] \xleftarrow{T,gc,2} A[(\texttt{letrec}\ E\ \texttt{in}\ (\lambda X.S))\ S']$$

$$A[(\lambda X.S)\ S'] \xrightarrow{SR,lbeta} A[\texttt{letrec}\ X = S'\ \texttt{in}\ S]$$

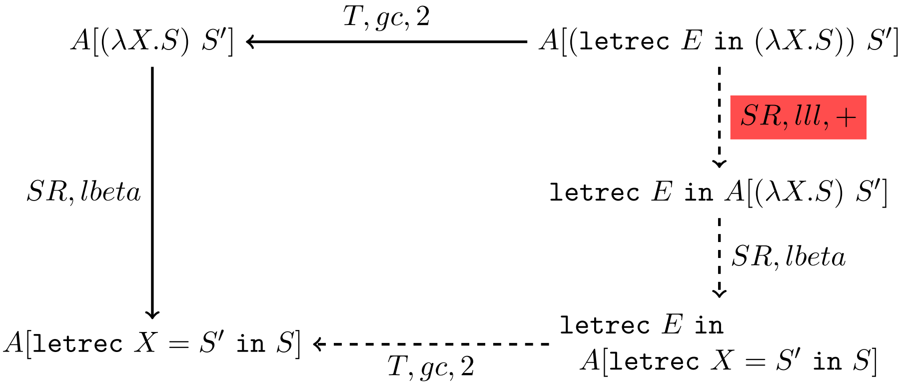
Example:



T, gc, 2
A[(λX.S) S′]
A[(letrec E in (λX.S)) S′]
SR, lll, +
SR, lbeta
letrec E in A[(λX.S) S′]
A[letrec X = S′ in S]

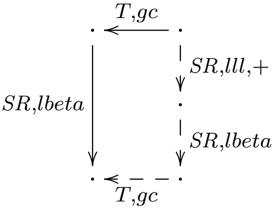
Transitive Closures are Required

Example:



Encoding of Transitive Closures

The diagram



is encoded by:

```
Tgc(SRlbeta(x)) -> gen(k,x)
gen(s(k),x) -> SRlll(gen(k,x))
gen(s(k),x) -> SRlll(SRlbeta(Tgc(x)))
```

- 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/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

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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

Further work

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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 α-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!