GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Unification of Program Expressions with Recursive Bindings

Manfred Schmidt-Schauß and David Sabel[†]

Goethe-University Frankfurt am Main, Germany

PPDP 2016, Edinburgh, UK

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

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Unification as a core procedure for

- **automated reasoning on programs and program transformations** w.r.t. operational semantics
- for program calculi with higher-order constructs and recursive bindings, e.g.

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

- special focus: extended call-by-need lambda calculi with letrec that model core languages of **lazy functional programming languages** like Haskell

Application: Correctness of Program Transformations

Program transformation T is correct iff $\forall \ell \to r \in T$: $\forall C$: $C[\ell]\downarrow \iff C[r]\downarrow$
where $\downarrow =$ successful evaluation w.r.t. standard reduction

Diagram-based proof method to show correctness of transformations:

- Compute **overlaps** between **standard reductions** and **program transformations** (automatable by **unification**)
- Join the overlaps $\Rightarrow$ forking and commuting diagrams
- Induction using the diagrams (automatable, see [RSSS12, IJCAR])

$$\begin{tikzcd}
\sigma(\ell_1)=\sigma(\ell_2) \arrow[r, "\text{program transformation}"] \arrow[d, "\text{standard reduction}"'] & \sigma(r_2) \arrow[d, dashed, "*"] \\
\sigma(r_1) \arrow[r, dashed, "*"'] & \cdot
\end{tikzcd}$$

unifier σ for $\ell_1 \doteq \ell_2$

$$\begin{tikzcd}
\sigma(\ell_1)=\sigma(r_2) \arrow[d, "\text{standard reduction}"'] & \sigma(\ell_2) \arrow[l, "\text{program transformation}"'] \arrow[d, dashed, "*"] \\
\sigma(r_1) & \cdot \arrow[l, dashed, "*"]
\end{tikzcd}$$

unifier σ for $\ell_1 \doteq r_2$

Design Decisions for 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)	$R[(\lambda x.e_1)\ e_2] \to R[\texttt{letrec}\ x = e_2\ \texttt{in}\ e_1]$
(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)	$T[\texttt{letrec}\ Env\ \texttt{in}\ e] \ \to\ T[e]$ if $LetVars(Env) \cap FV(e) = \emptyset$

Design Decisions for 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)	$R[(\lambda x.e_1)\ e_2] \to R[\texttt{letrec}\ x = e_2\ \texttt{in}\ e_1]$
(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)	$T[\texttt{letrec}\ Env\ \texttt{in}\ e] \ \to\ T[e]$ if $LetVars(Env) \cap FV(e) = \emptyset$

Meta-syntax must be capable to represent:

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

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

Meta-syntax must be capable to represent:

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

Design Decisions for the Meta-Syntax

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

Reduction contexts:

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

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

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)	$\texttt{letrec}\ \mathit{Env}_1\ \texttt{in}\ \texttt{letrec}\ \mathit{Env}_2\ \texttt{in}\ e \to \texttt{letrec}\ \mathit{Env}_1, \mathit{Env}_2\ \texttt{in}\ e$
(T,cpx)	$T[\texttt{letrec}\ x = y, \mathit{Env}\ \texttt{in}\ C[x]] \to T[\texttt{letrec}\ x = y, \mathit{Env}\ \texttt{in}\ C[y]]$
(T,gc)	$T[\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ e] \to T[e]$ if $\mathit{LetVars}(\mathit{Env}) \cap \mathit{FV}(e) = \emptyset$

Meta-syntax must be capable to represent:

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

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

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

Meta-syntax must be capable to represent:

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

Syntax of the Meta-Language LRSX

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

$$
\begin{array}{llcll}
\text{Variables} & x \in \textbf{Var} & ::= & X & \text{(variable meta-variable)} \\
 & & \mid & \mathsf{x} & \text{(concrete variable)} \\
\text{Expressions} & s \in \textbf{Expr} & ::= & S & \text{(expression meta-variable)} \\
 & & \mid & D[s] & \text{(context meta-variable)} \\
 & & \mid & \texttt{letrec}\ env\ \texttt{in}\ s & \text{(letrec-expression)} \\
 & & \mid & \texttt{var}\ x & \text{(variable)} \\
 & & \mid & (f\ r_1 \ldots r_{ar(f)}) & \text{(function application)} \\
 & & & \multicolumn{2}{l}{\quad \text{where } r_i \text{ is } o_i, s_i, \text{ or } x_i \text{ specified by } f} \\
 & o \in \textbf{HExpr}^n & ::= & x_1. \ldots .x_n.s & \text{(higher-order expression)} \\
\text{Environments} & env \in \textbf{Env} & ::= & \emptyset & \text{(empty environment)} \\
 & & \mid & E; env & \text{(environment meta-variable)} \\
 & & \mid & Ch[x, s]; env & \text{(chain meta-variable)} \\
 & & \mid & x.s; env & \text{(binding)}
\end{array}
$$

Binding and Scoping Constraints

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

Reduction contexts:

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

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

Standard-reduction rules and some program transformations

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

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

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

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

Unification-problems have to treat (implicit) restrictions on scoping and emptiness, e.g.:

- (gc): Env must not be empty; side condition on variables,
- (llet): $FV(Env_1) \cap LetVars(Env_2) = \emptyset$
- (cpx): x, y are not captured by C in $C[x]$

Binding and Scoping Constraints

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

Unification-problems have to treat (implicit) restrictions on scoping and emptiness, e.g.:

- (gc): Env must not be empty; side condition on variables,
- (llet): $FV(Env_1) \cap LetVars(Env_2) = \emptyset$
- (cpx): x, y are not captured by C in $C[x]$

Binding and Scoping Constraints

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

Unification-problems have to treat (implicit) restrictions on scoping and emptiness, e.g.:

- (gc): Env must not be empty; side condition on variables,
- (llet): $FV(Env_1) \cap LetVars(Env_2) = \emptyset$
- (cpx): x, y are not captured by C in $C[x]$

Binding and Scoping Constraints

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

Unification-problems have to treat (implicit) restrictions on scoping and emptiness, e.g.:

- (gc): Env must not be empty; side condition on variables,
- (llet): $FV(Env_1) \cap LetVars(Env_2) = \emptyset$
- (cpx): x, y are not captured by C in $C[x]$

Letrec Unification Problem

A letrec unification problem is a tuple $P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)$ with

- Γ: **unification equations** $s \doteq s'$
- Δ_1: **non-empty contexts** (set of D-variables)
- Δ_2: **non-empty environments** (set of E-variables)
- Δ_3: **non-capture constraints** (set of (expression,context)-pairs)

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 $P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)$

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

- $\rho(s) \sim_{let} \rho(s')$ for all $s \doteq s' \in \Gamma$
- $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$ and $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$
- $Var(\rho(s)) \cap CV(\rho(d)) = \emptyset$ for all $(s, d) \in \Delta_3$

A unifier ρ is a **solution of P** if ρ is a ground substitution.

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

$CV(d)$ = variables that are captured by the hole of context d

Solutions and Unifiers

Unifier and Solution of $P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)$

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

- $\rho(s) \sim_{let} \rho(s')$ for all $s \doteq s' \in \Gamma$
- $\rho(D) \neq [\cdot]$ for all $D \in \Delta_1$ and $\rho(E) \neq \emptyset$ for all $E \in \Delta_2$
- $Var(\rho(s)) \cap CV(\rho(d)) = \emptyset$ for all $(s, d) \in \Delta_3$

A unifier ρ is a **solution of P** if ρ is a ground substitution.

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

$CV(d)$ = variables that are captured by the hole of context d

Theorem (NP-Hardness)

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

Proof by a reduction from $\textsc{Monotone one-in-three-3-SAT}$.

Unification Algorithm UnifLRS

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

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 $(Sol, \emptyset, \Delta)$

Inference rules of the form $\dfrac{P}{P_1 \mid \ldots \mid P_n}$

Four kinds of rules:

- First-order rules
- Rules for environment equations
- Rules for equations $D[s] \doteq s'$
- Failure rules

Rule application is non-deterministic:

- don't care non-determinsm between the rules
- don't know non-determinism between $P_1 \mid \ldots \mid P_n$

Selection of Rules (1)

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

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

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

Unifying bindings and chains:

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

Keep chain-equations as constraints

$$\frac{(Sol, \Gamma \mathbin{\dot\cup} \{E_1; \ldots; E_n \doteq Ch[y,s]\}, (\Delta_1, \Delta_2, \Delta_3, \Delta_4))}{(Sol, \Gamma, (\Delta_1, \Delta_2, \Delta_3, \Delta_4 \cup \{E_1; \ldots; E_n \doteq Ch[y,s]\}))}$$

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:

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

For a final state $(Sol, \emptyset, \Delta)$ satisfiabilty of Δ_4 is checked:
Guess an instantiation σ for all $E_1; \ldots; E_n \doteq Ch[y, s] \in \Delta_4$ s.t.

- $\sigma(Ch[y, s]) = y.D_1[Y_1]; Y_1.D_2[Y_2]; \ldots; Y_k.D_{k+1}[s]$
- $\sigma(E_i) \subseteq \{y.D_1[Y_1];\ Y_1.D_2[Y_2]; \ldots;\ Y_k.D_{k+1}[s]\}$ and $\sigma(E_i) \neq \emptyset$ if $E_i \in \Delta_2$
- $\sigma(E_1; \ldots; E_n) \sim_{let} \sigma(Ch[y, s])$
- all non-capture constraints in $\Delta_3\sigma$ are valid

Deliver Fail if no such instantiation exists.

Satisfiability Check of Constraint Equations

For a final state $(Sol, \emptyset, \Delta)$ satisfiabilty of Δ_4 is checked:
Guess an instantiation σ for all $E_1; \ldots; E_n \doteq Ch[y, s] \in \Delta_4$ s.t.

- $\sigma(Ch[y, s]) = y.D_1[Y_1]; Y_1.D_2[Y_2]; \ldots; Y_k.D_{k+1}[s]$
- $\sigma(E_i) \subseteq \{y.D_1[Y_1];\ Y_1.D_2[Y_2]; \ldots;\ Y_k.D_{k+1}[s]\}$ and $\sigma(E_i) \neq \emptyset$ if $E_i \in \Delta_2$
- $\sigma(E_1; \ldots; E_n) \sim_{let} \sigma(Ch[y, s])$
- all non-capture constraints in $\Delta_3\sigma$ are valid

Deliver Fail if no such instantiation exists.

Key Lemma

It suffices to test only those k with $k + 1 \leq {M_1}^2 * (M_2 + 1) + M_2$ where $M_1 = |\Delta_2 \cap \{E_1; \ldots; E_n\}|$ and $M_2 = n - M_1$.
Thus, the Δ_4-check can be done in nondeterministic polynomial time.

Soundness and Completeness

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.

Theorem

UnifLRS **terminates in nondeterministic polynomial time** and solutions are of polynomial size.

Corollary

The letrec unification problem is NP-complete.

Computing Overlaps with UnifLRS

Implementation available from http://goethe.link/lrsx

- unification of expressions
- calculus descriptions as input for computing overlaps

Experiments with two call-by-need calculi:

- L_{need}: lambda calculus plus `letrec`
- LR: L_{need} + data constructors + case expressions + seq-expressions
- overlaps for 11 transformations w.r.t. all standard reduction rules

Statistics:

	Calculus L_{need}		Calculus LR	
number of standard rules	13		76	
	forking	commuting	forking	commuting
number of critical pairs	1741	2156	34319	37016
execution time (sec.)	2	3	44	56

Conclusion

- Sound and complete unification algorithm for program calculi with recursive bindings
- Letrec unification problem is NP-complete
- Automated computation of overlaps for call-by-need core languages is possible

- Sound and complete unification algorithm for program calculi with recursive bindings
- Letrec unification problem is NP-complete
- Automated computation of overlaps for call-by-need core languages is possible

Further work:

- Join the critical pairs: Requires matching-algorithm, but also handling of the $(\Delta_1, \Delta_2, \Delta_3, \Delta_4)$-constraints, and probably some kind of meta alpha-renaming
- Equivalence of different reductions strategies: computing overlaps requires to unify chain-variables $(Ch_1[y, s] \doteq Ch_2[y', s'])$