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.
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.

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

- special focus: extended call-by-need lambda calculi with letrec that model core languages of lazy functional programming languages like Haskell
Program transformation $T$ is **correct** iff $\forall \ell \rightarrow r \in T: \forall C: C[\ell] \downarrow \iff C[r] \downarrow$
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])
**Operational semantics of typical call-by-need calculi (excerpt)**

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

Standard-reduction rules and some program transformations

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

(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) \[ T[\text{letrec } Env \text{ in } e] \rightarrow T[e] \text{ if } \text{LetVars}(Env) \cap \text{FV}(e) = \emptyset \]
Operational semantics of typical call-by-need calculi (excerpt)

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

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

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

Standard-reduction rules and some program transformations

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

(SR,\!l\!let) \quad \text{letrec } Env_1 \text{ in letrec } Env_2 \text{ in } e \rightarrow \text{letrec } Env_1, Env_2 \text{ in } e

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

(T,\!gc) \quad T[\text{letrec } Env \text{ in } e] \rightarrow 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 \( Env_i \),
- environment chains \( \{x_i = A_i[x_{i+1}]\}_{i=1}^{n-1} \)
Operational semantics of typical call-by-need calculi (excerpt)

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

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

Meta-syntax must be capable to represent:

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

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) \[ \text{letrec Env}_1 \, \text{in letrec Env}_2 \, \text{in } e \rightarrow \text{letrec Env}_1, \text{Env}_2 \, \text{in } e \]
- (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) \[ T[\text{letrec Env in } e] \rightarrow T[e] \quad \text{if } \text{LetVars(Env)} \cap \text{FV}(e) = \emptyset \]

Meta-syntax must be capable to represent:

- contexts of different classes
- environments \( \text{Env}_i \),
- 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)

- \text{letrec } env \text{ in } s$
  - (letrec-expression)

- \text{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 $f$

### Environments

$env \in \text{Env} ::= \emptyset$

- (empty environment)

- $E; env$
  - (environment meta-variable)

- $Ch[x, s]; env$
  - (chain meta-variable)

- $x.s; env$
  - (binding)
Operational semantics of typical call-by-need calculi (excerpt)

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

Standard-reduction rules and some program transformations

(SR,\beta) \quad R[(\lambda x.e_1) e_2] \to R[\text{letrec } x = e_2 \ in \ e_1]
(SR,\lambda l) \quad \text{letrec } Env_1 \ in \ \text{letrec } Env_2 \ in \ e \to \text{letrec } Env_1, Env_2 \ in \ e
(T,\text{cpx}) \quad T[\text{letrec } x = y, Env \ in \ C[x]] \to T[\text{letrec } x = y, Env \ in \ C[y]]
(T,\text{gc}) \quad T[\text{letrec } Env \ in \ e] \to T[e] \quad \text{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): \( FV(Env_1) \cap \text{LetVars}(Env_2) = \emptyset \)
- (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:
\[ A ::= \cdot \mid (A \ e) \]
\[ R ::= A \mid \text{letrec } Env \ \text{in } A \mid \text{letrec } \{x_1=A_i[x_{i+1}]\}_{i=1}^{n-1}, x_n=A_n, Env, \text{in } A[x_1] \]

Standard-reduction rules and some program transformations

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

(SR,\lambda\letrec) \quad \text{letrec } Env_1 \ \text{in } \text{letrec } Env_2 \ \text{in } e \to \text{letrec } Env_1, Env_2 \ \text{in } e

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

(T,\text{gc}) \quad T[\text{letrec } Env \ \text{in } e] \to T[e] \quad \text{if } \text{LetVars}(Env) \cap \text{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): \( \text{FV}(Env_1) \cap \text{LetVars}(Env_2) = \emptyset \)
- (cpx): \( x, y \) are not captured by \( C \) in \( C[x] \)
Operational semantics of typical call-by-need calculi (excerpt)

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

Standard-reduction rules and some program transformations

(SR,\text{Ibeta}) \ R[(\lambda x. e_1) \ e_2] \rightarrow R[\text{letrec } x = e_2 \text{ in } e_1]
(SR,\text{Ilet}) \ \text{letrec } Env_1 \text{ in letrec } Env_2 \text{ in } e \rightarrow \text{letrec } Env_1, Env_2 \text{ in } e
(T,\text{cpx}) \ T[\text{letrec } x = y, Env \text{ in } C[x]] \rightarrow T[\text{letrec } x = y, Env \text{ in } C[y]]
(T,\text{gc}) \ T[\text{letrec } Env \text{ in } e] \rightarrow T[e] \quad \text{if LetVars}(Env) \cap \text{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,
- (Ilet): \( \text{FV}(Env_1) \cap \text{LetVars}(Env_2) = \emptyset \)
- (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:
\[ A ::= \cdot | (A \, e) \]
\[ R ::= A \mid \text{letrec } Env \, \text{in } A \mid \text{letrec } \{ x_i = A_i[x_{i+1}] \}_{i=1}^{n-1}, x_n = A_n, Env, \, \text{in } A[x_1] \]

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) \[ \text{letrec } Env_1 \, \text{in } \text{letrec } Env_2 \, \text{in } e \rightarrow \text{letrec } Env_1, Env_2 \, \text{in } e \]
(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) \[ T[\text{letrec } Env \, \text{in } e] \rightarrow T[e] \quad \text{if } \text{LetVars}(Env) \cap \text{FV}(e) = \emptyset \]

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

- (gc): \( Env \) must not be empty; side condition on variables,
- (llet): \( \text{FV}(Env_1) \cap \text{LetVars}(Env_2) = \emptyset \)
- (cpx): \( x, y \) are not captured by \( C \) in \( C[x] \)
A letrec unification problem is a tuple $P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)$ with

- $\Gamma$: unification equations $s \equiv s'$
- $\Delta_1$: non-empty contexts (set of $D$-variables)
- $\Delta_2$: non-empty environments (set of $E$-variables)
- $\Delta_3$: non-capture constraints (set of (expression,context)-pairs)

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

A substitution $\rho$ is a \textbf{unifier of} $P$ iff

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

A unifier $\rho$ is a \textbf{solution of} $P$ if $\rho$ is a ground substitution.

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

$\text{CV}(d) =$ variables that are captured by the hole of context $d$
Unifier and Solution of $P = (\Gamma, \Delta_1, \Delta_2, \Delta_3)$

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

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

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

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

$\text{CV}(d) = \text{variables that are captured by the hole of context } d$

**Theorem (NP-Hardness)**

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

Proof by a reduction from **Monotone one-in-three-3-SAT**.
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 \((Sol, \emptyset, \Delta)\)
Inference rules of the form

$$P \quad \frac{P_1 | \ldots | P_n}{\quad}$$

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-determinism between the rules
- don’t know non-determinism between $P_1 | \ldots | P_n$
Selection of Rules (1)

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

\[ (\text{Sol}, \Gamma, \Delta) \]

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

\[ (\text{Sol} \circ \{ S \mapsto s \}, \Gamma[s/S], \Delta[s/S]) \]

if \( S \) is not a proper sub-expression of \( s \)

\[ (\text{Sol}, \Gamma \cup \{ \text{letrec } \text{env}_1 \text{ in } s_1 \doteq \text{letrec } \text{env}_2 \text{ in } s_2 \}, \Delta) \]

\[ (\text{Sol}, \Gamma \cup \{ \text{env}_1 \doteq \text{env}_2, s_1 \doteq s_2 \}, \Delta) \]
Unifying bindings and chains:

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

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

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

“equal”

\[(Sol \circ \sigma, \Gamma \cup \{x.t \models y.D[\text{var} Y], env_1 \models Ch_2[Y, s]; env_2\}, \Delta_\sigma)\]

\[\sigma = \{Ch_1[y, s] \mapsto y.D[\text{var} Y]; Ch_2[Y, s]\}\]

“prefix”

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

\[\sigma = \{Ch[y, s] \mapsto Ch_1[y, (\text{var} Y_1)]; Y_1.D[\text{var} Y_2]; Ch_2[Y_2, s]\}\]

“infix”

\[(Sol \circ \sigma, \Gamma \cup \{x.t \models Y_1.D[s], env_1 \models Ch_2[y, \text{var} Y_1]; env_2, \Delta_\sigma\})\]

\[\sigma = \{Ch_1[y, s] \mapsto Ch_2[y, \text{var} Y_1]; Y_1.D[s]\}\]

“suffix”
Keep chain-equations as constraints

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

$$(\text{Sol}, \Gamma, (\Delta_1, \Delta_2, \Delta_3, \Delta_4 \cup \{E_1; \ldots; E_n \doteq Ch[y, s]\}))$$
Selection of Failure Rules

Standard cases:

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

\[
(Sol, \Gamma \cup \{(S \equiv s)\}, \Delta) \\
Fail \quad \text{if } S \text{ is a proper subterm of } s
\]

Checking non-capture constraints:

\[
(Sol, \Gamma, (\Delta_1, \Delta_2, \Delta_3 \cup \{(s, d)\}, \Delta_4)) \\
Fail \quad \text{if } \text{Var}(s) \cap \text{CV}(d) \neq \emptyset
\]
For a final state \((Sol, \emptyset, \Delta)\) satisfiability of \(\Delta_4\) is checked:

**Guess** an instantiation \(\sigma\) for all \(E_1; \ldots; E_n \vdash Ch[y, s] \in \Delta_4\) 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 \(E_i \in \Delta_2\)
- \(\sigma(E_1; \ldots; E_n) \sim_{\text{let}} \sigma(Ch[y, s])\)
- 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 \((Sol, \emptyset, \Delta)\) satisfiability of \(\Delta_4\) is checked:

**Guess** an instantiation \(\sigma\) for all \(E_1; \ldots; E_n \vdash Ch[y, s] \in \Delta_4\) 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 \(E_i \in \Delta_2\)
- \(\sigma(E_1; \ldots; E_n) \sim \text{let } \sigma(Ch[y, s])\)
- 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 \(k + 1 \leq M_1^2 \ast (M_2 + 1) + M_2\)
where \(M_1 = |\Delta_2 \cap \{E_1; \ldots; E_n\}|\) and \(M_2 = n - M_1\).
Thus, the \(\Delta_4\)-check can be done in nondeterministic polynomial time.
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.
Complexity of UnifLRS

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:

<table>
<thead>
<tr>
<th></th>
<th>Calculus ( L_{need} )</th>
<th>Calculus LR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>forking</td>
<td>commuting</td>
</tr>
<tr>
<td>number of standard rules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number of critical pairs</td>
<td>1741</td>
<td>2156</td>
</tr>
<tr>
<td>execution time (sec.)</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
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
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

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
  
  \( (Ch_1[y, s] \equiv Ch_2[y', s']) \)