Implementing Rigid E-Unification (extended abstract)

Michael Franssen
UNIF2009, August 2\textsuperscript{nd} 2009
Overview

Theory:
• BSE calculus
• Solved forms
• Simple systems

Implementation
• eBSE calculus
• Solved forms
• Simple systems graph

Results
The BSE calculus (Degtyarev and Voronkov)

Rigid E-unification: Given a set $E$ of equations, see if $s = t$ can be derived from $E$ (i.e. $E \vdash s = t$)

\[
\begin{align*}
\text{lrb}\! &\! s \quad E \cup \{ l = r, s[p] = t \} \vdash e \cdot C \\
\frac{E \cup \{ l = r, s[r] = t \} \vdash e \cdot C \land l \succ r \land s[p] \succ t \land l \succeq p}{p \not\in \mathcal{V}, s[r] \neq t}
\end{align*}
\]

\[
\begin{align*}
\text{rrb}\! &\! s \quad E \cup \{ l = r \} \vdash s[p] = t \cdot C \\
\frac{E \cup \{ l = r \} \vdash s[r] = t \cdot C \land l \succ r \land s[p] \succ t \land l \succeq p}{p \not\in \mathcal{V}}
\end{align*}
\]

\[
\begin{align*}
\text{er} \quad E \vdash s = t \cdot C \\
\frac{E \vdash s = t \cdot C}{s \not= t}
\end{align*}
\]

The ordering constraint in the BSE rules is used to ensure termination (i.e. terms become smaller)

The real problem is to compute satisfiability of the constraint
Comon introduced a rewrite system $R$ to simplify the ordering constraint into a set of solved forms:

$$x_1 ≃ t_1 \land \ldots \land x_n ≃ t_n \land u_1 ≻ v_1 \land \ldots \land u_m ≻ v_m$$

Each solved form explicitly states for every rigid variable a value, an upper bound or a lower bound.

For example, $R$ contains the rules:

**(R)** \( x ≃ t \land P \rightarrow_R x ≃ t \land P[x:=t] \) if \( x \in \text{FV}(P) \setminus \text{FV}(t) \), \( P \) is a conjunction of (in)equations and \( t \in V \Rightarrow t \in \text{FV}(P) \)

**(D2)** \( f(v_1,\ldots,v_n) ≻ g(u_1,\ldots,u_m) \) if \( f \succ_f g \)

\[ \rightarrow_R f(v_1,\ldots,v_n) ≻ u_1 \land \ldots \land f(v_1,\ldots,v_n) ≻ u_m \]
Simple systems (Comon and Nieuwenhuis)

The equations are part of the solution. The inequations constitute a constrained part.

To check if the inequations are satisfiable, we need to construct the corresponding simple systems:

- For this, consider all orderings of all subterms of a constrained part, where every subterm $s_1$ of $s_2$ occurs after $s_2$.
- Put $\approx$ or $>$ between these subterms in all possible ways that are compatible with the constrained part.

For every simple system constructed this way, check if it is satisfiable.

(this is omitted in this talk, but fairly trivial)
A small example

Consider a constrained part \( f(g(x)) \succ y \).

The set of subterms is \( \{f(g(x)), g(x), x, y\} \).

The possible orderings are:
1. \( f(g(x)), g(x), x, y \)
2. \( f(g(x)), g(x), y, x \)
3. \( f(g(x)), y, g(x), x \)
4. \( y, f(g(x)), g(x), x \)

The 17 simple systems are:

\[
\begin{align*}
\text{f(g(x))} & \equiv \text{g(x)} \equiv x \succ y \\
\text{f(g(x))} & \equiv \text{g(x)} \equiv y \equiv x \\
\text{f(g(x))} & \equiv \text{g(x)} \equiv x \equiv y \\
\text{f(g(x))} & \equiv \text{g(x)} \equiv y \equiv x \\
\text{f(g(x))} & \equiv \text{g(x)} \equiv x \equiv y \\
\text{f(g(x))} & \equiv \text{g(x)} \equiv y \equiv x \\
\end{align*}
\]
Implementation

To obtain an efficient implementation, we will

• modify the \textit{BSE} calculus into the \textit{eBSE} calculus to allow pre-computation and caching of sub-results.

• efficiently implement the rewrite system $R$, without storing sub-results or actually performing substitutions.

• use a graph structure to construct as little \textit{simple systems} as possible and cut of the construction as soon as possible
The eBSE calculus maintains a constraint for each equation. No equations are removed from the left hand side. Hence, the closure $E'$ of $E$ w.r.t. $lrbs$ can be pre-computed and re-used. When $E$ is extended, the new closure is represented by an equation list, whose tail is the old closure. Hence, every equation is stored (and computed) only once.
Computing solved forms

A substitution $\theta$ is represented by a list of one-point substitutions:

$$\theta' = <[x_1 \mapsto t_1], \ldots, [x_n \mapsto t_n]>$$

The head symbol of $\theta(e)$ can be computed by repeatedly using the one-point mappings in $\theta'$. $\theta$ itself is not explicitly represented or applied.

*Solved* is a datatype consisting of the solved part $\theta'$ and a list of inequations representing the constrained part.

The idea is to implement functions like

{Solved} makeGreater(S:Solved, A:Term, B:Term),

which recursively compute all possible extensions of $S$ in which $\theta(A) > \theta(B)$ will hold, where $\theta$ is the solved part of $S$.

These functions recursively unfold $R$ on a depth first basis, so intermediate results are never stored.

All one-point substitutions of $\theta'$ and all terms in the constrained part already exist in the original problem, hence no copies are needed.
Computing a satisfiable simple system

To compute a satisfiable simple system we first build a graph with all subterms that is compatible with LPO and the original constraint.

The graph is then recursively traversed to construct all possible simple systems. Satisfiability of the simple system is computed on the fly and recursion is aborted as soon as satisfiability fails.

The graph for $f(g(x)) \succ y$ becomes:

- Generating:
  - $f(g(x)) \succ g(x) \succ x \approx y$
  - $f(g(x)) \succ g(x) \succ y \approx x$
  - $f(g(x)) \succ y \approx g(x) \succ x$
  - $f(g(x)) \succ g(x) \succ x \succ y$
  - $f(g(x)) \succ g(x) \approx y \succ x$
  - $f(g(x)) \succ y \succ g(x) \succ x$
  - $f(g(x)) \succ g(x) \succ y \approx x$

- Or even just:
  - $f(g(x)) \succ g(x) \succ y \approx x$
  - $f(g(x)) \succ y \approx g(x) \succ x$
  - $f(g(x)) \succ g(x) \succ x \succ y$
  - $f(g(x)) \succ y \succ g(x) \succ x$
  - $f(g(x)) \succ g(x) \succ y \succ x$
Results

We have a full implementation of rigid E-unification, which can relatively easy be integrated in an automated theorem prover.

The eBSE calculus allows for pre-computing applications of the \textit{lrbs} rule.

Solved forms are computed without performing any substitutions and without storing intermediate results.

The simple systems graph avoids the creation of most unsatisfiable simple systems, by systematically creating only those conforming to LPO and checking satisfiability on the fly.

It is now integrated in Cocktail's automated tableaux prover!
Questions?