A Functional Implementation of Function-as-Constructor Higher-Order Unification

Makoto Hamana

Department of Computer Science,
Gunma University, Japan

September 3, UNIF 2017, Oxford

This Work

- [Libal and Miller FSCD'16]: A new decidable class of higher-order unification problems, Functions-as-Constructors unification (FCU)
- Report here that FCU unification can be implemented functionally
- SOL: Haskell-based tool for analysing confluence and termination of second-order computatoin rules
  - HO version of Knuth and Bendix's critical pair checking using FCU unification

Example: Computation Rules on Sum types

case	$: \boldsymbol{a_1 + a_2, (a_1 \longrightarrow c), (a_2 \longrightarrow c) \longrightarrow c}$
inl	$: \boldsymbol{a_1 \longrightarrow a_1 + a_2}$ inr $: \boldsymbol{a_2 \longrightarrow a_1 + a_2}$

(caseL)	case(inl$(\boldsymbol{X}), \boldsymbol{F}, \boldsymbol{G})$	$\Rightarrow \boldsymbol{F(X)}$
(caseR)	case(inr$(\boldsymbol{Y}), \boldsymbol{F}, \boldsymbol{G})$	$\Rightarrow \boldsymbol{G(Y)}$
(sumEta)	case$(\boldsymbol{Z}, \boldsymbol{\lambda x.H}[\mathsf{inl}(\boldsymbol{x})], \boldsymbol{\lambda y.H}[\mathsf{inr}(\boldsymbol{y})])$	$\Rightarrow \boldsymbol{H}[\boldsymbol{Z}]$

$\boldsymbol{\lambda x.H}[\mathsf{inl}(\boldsymbol{x})]$, $\boldsymbol{\lambda x.H}[\mathsf{inr}(\boldsymbol{y})]$ are not higher-order patterns

Higher-Order Patterns [Miller'91]

- A higher-order pattern
  is a term where every application is of the form $M[x_1, \ldots, x_n]$
  i.e.
  a free variable M applied to distinct bound variables $x_1, \ldots, x_n$.

- Not HO patterns
  - $M[N]$
  - $M[\mathrm{cons}(x, y)]$
  - $\lambda x.H[\mathrm{inl}(x)]$
  - $\lambda y.H[\mathrm{inr}(y)]$

FC patterns [Yokoyama et al. '04][Libal, Miller '17]

▷ An FC pattern is a term p,
where every occurrence of $M[t_1, \ldots, t_n]$ in p:

(i) every t_i is a term without binders or free variables,
can contain fun. syms with arity $n > 0$ and bound variables

(ii) every t_i contains at least one bound variable,

(iii) $t_i \not\trianglelefteq t_j$ for every $1 \leq i, j \leq n$.

▷ FC patterns:

$\lambda x.y.M[\mathsf{cons}(x, y)]$ $\lambda x.H[\mathsf{inl}(x)]$ $\lambda y.H[\mathsf{inr}(y)]$

▷ Not FC patterns:

$M[N]$ $\lambda x.M[x, x]$ $M[\mathsf{nil}]$ $\lambda x.M[x, c(x)]$

FC Pattern Matching and Notice

Thm. [Yokoyama et al. Inf. Process. Lett.'04]

Any second-order FC pattern matching problem $\boldsymbol{p} \stackrel{?}{=} \boldsymbol{t}$ between an FU pattern $\boldsymbol{p}$ and a $\boldsymbol{\lambda}$-term $\boldsymbol{t}$ is decidable and has a single most general matcher if matchable.

But the unification between FC patterns

$$\boldsymbol{x.y.M}[\mathsf{c}(\boldsymbol{x}), \mathsf{c}(\boldsymbol{y})] \stackrel{?}{=} \boldsymbol{x.y.}\mathsf{c}(\boldsymbol{N}[\boldsymbol{y}, \boldsymbol{x}])$$

has at least two incomparable unifiers:

$$\{\boldsymbol{M} \mapsto \boldsymbol{x.y.y},\ \boldsymbol{N} \mapsto \boldsymbol{x.y.x}\} \text{ and } \{\boldsymbol{M} \mapsto \boldsymbol{x.y.x},\ \boldsymbol{N} \mapsto \boldsymbol{x.y.y}\}.$$

FCU Unification

- Libal and Miller' **Functions-as-Constructors Unification (FCU)** [FSCD'16]

- A FCU unification problem is $s \stackrel{?}{=} t$ where s and t are FC patterns and satisfies:
  - **Global restriction:** in $s \stackrel{?}{=} t$, for every two different occurrences of applications $M[s_1, \ldots, s_n]$ and $N[t_1, \ldots, t_m]$, $s_i \not\trianglelefteq t_j$ holds

- **Thm.** An FCU unification problem is decidable and ensures the existence of a most general unifier if solvable.

- Yokoyama et al.'s example actually violates the global restriction

$$x.y.M[\mathsf{c}(x), \mathsf{c}(y)] \stackrel{?}{=} x.y.\mathsf{c}(N[y, x])$$

Implementation

- Written in Haskell (about 500 lines)
- as a part of SOL system
- Algorithm in [Libal,Miler'16] is not immediately ready
- Base Nipkow's ML implementation of pattern unification
  - a basic library and infrastructure for higher-order unification
  - e.g. on-the-fly α-conversion and η-expansion.

FCU Algorithm

a slight modification of Libal and Miller's, adapted to Nipkow's formalism

(idem) $Q_\forall \quad t \stackrel{?}{=} t \quad \longrightarrow \quad Q_\forall \quad []$

(abs) $Q_\forall \quad \lambda x.s \stackrel{?}{=} \lambda x.t \quad \longrightarrow \quad x, Q_\forall \quad s \stackrel{?}{=} t$

(fun) $Q_\forall \quad f\ \overrightarrow{s} \stackrel{?}{=} f\ \overrightarrow{t} \quad \longrightarrow \quad Q_\forall \quad s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n$

(flex-rigid) $Q_\forall \quad F\ \overrightarrow{t} \stackrel{?}{=} f\ \overrightarrow{s} \quad \longrightarrow \quad Q_\forall \quad []$

$\{F \mapsto \lambda \overrightarrow{z}.\text{discharge (zip } \overrightarrow{t}\ \overrightarrow{z})\ (f \overrightarrow{s})\}$

(flex-flex=) $Q_\forall \quad F\ \overrightarrow{t} \stackrel{?}{=} F\ \overrightarrow{s} \quad \longrightarrow \quad Q_\forall \quad []$

$\{X \mapsto \lambda z_1, \ldots, z_n.H \overrightarrow{z}'\}$

where $\overrightarrow{z}' = (z_i \mid 1 \leq i \leq n, t_i = s_i)$

(flex-flex≠) $Q_\forall \quad F\ \overrightarrow{t} \stackrel{?}{=} G\ \overrightarrow{s} \quad \longrightarrow \quad Q_\forall \quad s \stackrel{?}{=} t$

$\{Y \mapsto \lambda z_1, \ldots, z_m.H \overrightarrow{z_{\varphi(i)}}\}$,

where $\varphi(j) = i$ if $t_i = s_j$ for $i = 1, \ldots, n,\ j = 1, \ldots, m$.

Actual Transformation Relation

$$\langle Q_\forall,\ (s \overset{?}{=} t) : E,\ \theta \rangle \longrightarrow \langle Q'_\forall, E' +\!\!+ (E\theta')\!\downarrow_\beta,\ \theta' \circ \theta \rangle$$

if $\langle Q_\forall, s \overset{?}{=} t \rangle \longrightarrow \langle Q'_\forall, E', \theta' \rangle$

- $(-)\!\downarrow_\beta$ computes the β-normal form
- apply the “pruning” operation if applicable

Discharging

- ▷ Operation $\boldsymbol{t}|^{\vec{s}}_{\vec{z}}$ Yokoyama et al. [I. P. L. '04] called "discharging", gave a complicated algorithm
- ▷ Replace terms $\overrightarrow{\boldsymbol{s}}$ in $\boldsymbol{t}$ with variables $\overrightarrow{\boldsymbol{z}}$.
- ▷ Similar to substitution of terms for variables,
- ▷ Hence, implement $\boldsymbol{t}|^{\vec{s}}_{\vec{z}}$ as `discharge` $\boldsymbol{\theta}$ `t`

```
discharge :: [(Term, Id)] -> Term -> Term
discharge th t' = case lookup t' th of
  Just z  -> O z
  Nothing -> case t of
               (x :.: t1) -> x :.: discharge th t1
               (t1 :@ t2) -> (discharge th t1) :@ (discharge th t2)
               t'         -> t'
```

Unificatoin Function

```
unif bvs (th,(s,t))
```

processes a unificatoin problem $\langle Q_\forall,\ (s \stackrel{?}{=} t),\ \theta\rangle$.

```
unif :: [(Char,Id)] -> ([(Id, Term)], (Term, Term)) -> [(Id, Term)]
unif bvs (th,(s,t)) = case (devar th s,devar th t) of
        (x:.:s,y:.:t) -> unif (('B',x):bvs) (th,(s,if x==y then t
                                                   else rename x y t))
        (s,t)         -> cases bvs th (s,t)


cases bvs th (s,t) = case (strip s,strip t) of
 ((W _F,ym),(W _G,zn)) -> flexflex bvs (_F,ym,_G,zn,th)
 ((W _F,ym),_)         -> flexrigid bvs (_F,ym,t,th)
 (_,(W _F,ym))         -> flexrigid bvs (_F,ym,s,th)
 ((a,sm),(b,tn))       -> rigidrigid bvs (a,sm,b,tn,th)
```

Example: Computation Rules on Sum types

case $: \boldsymbol{a_1 + a_2, (a_1 \longrightarrow c), (a_2 \longrightarrow c) \longrightarrow c}$

inl $: \boldsymbol{a_1 \longrightarrow a_1 + a_2}$ inr $: \boldsymbol{a_2 \longrightarrow a_1 + a_2}$

(caseL) case(inl($\boldsymbol{X}$), $\boldsymbol{F}, \boldsymbol{G}$) $\Rightarrow \boldsymbol{F(X)}$

(caseR) case(inr($\boldsymbol{Y}$), $\boldsymbol{F}, \boldsymbol{G}$) $\Rightarrow \boldsymbol{G(Y)}$

(sumEta) case($\boldsymbol{Z}, \boldsymbol{\lambda x.H}$[inl($\boldsymbol{x}$)], $\boldsymbol{\lambda y.H}$[inr($\boldsymbol{y}$)]) $\Rightarrow \boldsymbol{H[Z]}$

Conclusion

- Report here that FCU unification can be implemented functionally
- SOL: Haskell-based tool for analysing confluence and termination of second-order computatoin rules
  - HO version of Knuth and Bendix's critical pair checking using FCU unification