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

Makoto Hamana[1]

Department of Computer Science, Gunma University, Japan
hamana@cs.gunma-u.ac.jp

Abstract

Recently, Libal and Miller showed a new decidable class of higher-order unification problems called Functions-as-Constructors unification (FCU). It extends the class of higher-order patterns and retains good properties of pattern unification. But no implementation of FCU unification has been given. We report here that FCU unification can be implemented nicely functionally. We have implemented the FCU unification in Haskell as a part of our system SOL, Second-Order Laboratory, for critical pair checking of higher-order rules. Our experience on SOL system analysing various computation rules in programming languages shows that the functional FCU unification is useful in practice.

1 Introduction

It is widely known that Miller's higher-order patterns provide a decidable class of higher-order unification [Mil91]. These have been used as the basic term structure for pattern matching in various computational systems, such as higher-order logic programming [Mil91] and rewrite rules [MN98]. Higher-order patterns have nice properties: there exists the most general unifier (*mgu*) for a solvable unification problem, and an efficient algorithm is known [Bax77]. However, the class is a bit restrictive in practice.

Consider the following example. Suppose a simple type system having sum types $a + b$ and the function symbols $\mathsf{case} : a_1 + a_2, (a_1 \to c), (a_2 \to c) \to c$ for case-construct and $\mathsf{inl} : a_1 \to a_1 + a_2$, $\mathsf{inr} : a_2 \to a_1 + a_2$ for inclusions. Consider the computation rules on sum types

$$\begin{array}{lll} \text{(caseL)} & \mathsf{case}(\mathsf{inl}(X), F, G) & \Rightarrow F(X) \\ \text{(caseR)} & \mathsf{case}(\mathsf{inr}(Y), F, G) & \Rightarrow G(Y) \\ \text{(sumEta)} & \mathsf{case}(Z, \lambda x.H[\mathsf{inl}(x)], \lambda y.H[\mathsf{inr}(y)]) & \Rightarrow H[Z] \end{array}$$

In this paper, we use the notation $M[t_1, \ldots, t_n]$ to denote a λ-term of application $M\ t_1\ \cdots\ t_n$ with a free variable M. The rules (caseL) and (caseR) define the computation behavior of the case-construct. The rule (sumEta) is the η-reduction rule of case-expression. An important fact is that $\lambda x.H[\mathsf{inl}(x)]$ and $\lambda y.H[\mathsf{inr}(y)]$ are *not higher-order patterns.*

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 must be applied to distinct bound variables $x_1, \ldots, x_n$. Therefore, terms such as $M[N], M[\mathsf{cons}(x, y)]$, $\lambda x.H[\mathsf{inl}(x)], \lambda y.H[\mathsf{inr}(y)]$ are not higher-order patterns. Hence, the existing algorithm for higher-order pattern matching cannot be used for computing with a rule such as (sumEta). This is problematic for general higher-order computation. Moreover, it means that we cannot compute overlapping between rules to compute *critical pairs* for local confluence [KB70]. It needs unification between the left-hand sides of rules of this kind of higher-order rules. However, the existing higher-order pattern unification cannot cope with the terms $\lambda x.H[\mathsf{inl}(x)]$ and $\lambda y.H[\mathsf{inr}(y)]$.

Recently, Libal and Miller present a new decidable class of higher-order unification problems called *Functions-as-Constructors unification (FCU)* [LM16]. It extends the class of higher-order pattern unification and retains good properties of pattern unification, namely, it ensures the existence of the most general unifier if an FCU problem is unifiable. Interestingly, it can solve the above mentioned problem. Terms such as $\lambda x.H[\mathsf{inl}(x)]$ and $\lambda y.H[\mathsf{inr}(y)]$ can be targets of FCU unification. Hence, we have recognised that FCU unification is useful to formulate a new effective class for higher-order computation [Ham17].

In this paper, we report that FCU unification can be implemented functionally. In [LM16], Libal and Miller gave an algorithm for FCU unification, but they did not provide its implementation. They mentioned that the implementation is a further work. We have implemented the FCU unification algorithm and its slight extension in Haskell, which has been a part of the SOL system for critical pair checking of computation rules [Ham17]. Our experience on SOL system analysing various computation rules in programming languages shows that the functional FCU unification is useful in practice.

2 FC patterns and matching

We first introduce the notion of FC patterns and matching. We denote by $s \trianglelefteq t$ if s is a subterm of t, and $s \triangleleft t$ if $s \trianglelefteq t$ and $s \neq t$. We use the notation $\overline{A}$ for a sequence $A_1, \cdots, A_n$.

An ***FC pattern*** is a term p in which every occurrence of application $M[t_1, \ldots, t_n]$ in p, the following conditions are satisfied:

(i) every t_i is a term without binders, metavariables or free variables, but it can contain function symbols 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 $1 \leq i, j \leq n$.

For example, $M[\mathsf{cons}(x, y)]$ is an FC pattern. FC patterns extend Miller's higher-order patterns, because "metavariables with distinct bound variables" are ensured by (ii)(iii). A *matching problem* $s \overset{?}{=} t$ is an equation consisting of terms s, t, both of which are in long β η-normal forms. It asks whether there exists a matcher θ such that $s\,\theta = t$ holds. A *unification problem* $s \overset{?}{=} t$ is an equation consisting higher-order patterns s, t in long β η-normal forms. It asks whether there exists a unifier θ such that $s\,\theta = t\,\theta$ holds.

Theorem 2.1. ([YHT04a]) *Any second-order FC pattern matching problem $p \overset{?}{=} t$ between an FU pattern p and a λ-term t is decidable and has a single most general matcher if matchable.*

What about unification? In contrast to matching, *unification* problem $p \overset{?}{=} t$ between FC patterns may not have a single most general *unifier*. Yokoyama et al. [YHT04b, Sec. 2] have shown such an example. The unification between FC patterns

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

has at least two incomparable unifiers: $\{M \mapsto x.y.y,\ N \mapsto x.y.x\}$ and $\{M \mapsto x.y.x,\ N \mapsto x.y.y\}$. FCU unification avoids this problem by identifying such a kind of problems.

3 FCU unification

Libal and Miller's *Functions-as-Constructors Unification (FCU)* [LM16] imposes an additional restriction to a unification problem in order to ensure the existence of mgu. A problem $s \overset{?}{=} t$ is

(idem)	$Q_\forall$	$t \overset{?}{=} t$	$\to$	$Q_\forall$	$[]$	$\varnothing$
(abs)	$Q_\forall$	$\lambda x.s \overset{?}{=} \lambda x.t$	$\to$	$x, Q_\forall$	$s \overset{?}{=} t$	$\varnothing$
(fun)	$Q_\forall$	$f\,\overline{s} \overset{?}{=} f\,\overline{t}$	$\to$	$Q_\forall$	$s_1 \overset{?}{=} t_1, \ldots, s_n \overset{?}{=} t_n$	$\varnothing$
(flex-rigid)	$Q_\forall$	$F\,\overline{t} \overset{?}{=} f\,\overline{s}$	$\to$	$Q_\forall$	$[]$	$\{F \mapsto \lambda\overline{z}.\mathsf{discharge}\ (\mathsf{zip}\ \overline{t}\ \overline{z})\ (f\overline{s})\}$
(flex-flex same)	$Q_\forall$	$F\,\overline{t} \overset{?}{=} F\,\overline{s}$	$\to$	$Q_\forall$	$[]$	$\{X \mapsto \lambda z_1, \ldots, z_n.H\overline{z}'\}$
		where $\overline{z}' = (z_i \mid 1 \le i \le n, t_i = s_i)$				
(flex-flex diff)	$Q_\forall$	$F\,\overline{t} \overset{?}{=} G\,\overline{s}$	$\to$	$Q_\forall$	$s \overset{?}{=} t$	$\{Y \mapsto \lambda z_1, \ldots, z_m.H\overline{z_{\varphi(i)}}\}$,
		where $\varphi(j) = i$ if $t_i = s_j$ for $i = 1, \ldots, n,\ j = 1, \ldots, m.$				

Figure 1: An FCU unification algorithm

called *FCU unification* if s and t are FC patterns and satisfies the following condition ([LM16, Def. 14]).

- **Global restriction:** in $s \overset{?}{=} t$, for every two different occurrences of applications $M[s_1, \ldots, s_n]$ and $N[t_1, \ldots, t_m]$, $s_i \ntrianglelefteq t_j$ holds for every $1 \le i \le n, 1 \le j \le m$.

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

Theorem 3.1. ([LM16]) *An FCU unification problem is decidable and ensures the existence of a most general unifier if solvable.*

3.1 Functional FCU unification

An FCU unification algorithm has been given in [LM16], which is shown to be terminating and returns a most general unifier. But no implementation of FCU unification has been reported in the literature. The algorithm in [LM16] is not immediately ready for implementation, because it requires several complex operation during unification such as *pruning* and *discharging*.

Nipkow has given a functional formulation of pattern unification and an ML implementation [Nip93]. Our strategy of implementing FCU is to base Nipkow's functional implementation and to adapt it to the case of FCU. This approach is solid and realistic because Nipkow's implementation has given a basic library and infrastructure for higher-order unification, such as on-the-fly α-conversion and η-expansion. Moreover, FCU was designed to be an improvement of pattern unification.

Based on this approach, we proceed to give a functional FCU. The transformation rules in Fig. 1, slightly modified from [LM16] for our functional implementation, are of the form

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

which is read as follows. The unification problem $s \overset{?}{=} t$ has a set of free variables (or, universally quantified by) $Q_\forall$ and it is translated to the list of unification problems E' under the new substitution θ'. In the rule (flex-rigid), the function $\mathsf{discharge}$ is a *discharging* function [YHT04a] that replaces every term t in $\overline{t}$ with a variable z in $\overline{z}$. In the rules (flex-flex same/diff), the variable H is assumed to be *fresh*. The actual transformation relation is defined by

$$\langle Q_\forall,\ (s \overset{?}{=} t) : E,\ \theta\rangle \longrightarrow \langle Q'_\forall, E' +\!\!+ (E\theta')\!\downarrow_\beta,\ \theta' \circ \theta\,\rangle \quad \text{if } \langle Q_\forall, s \overset{?}{=} t\rangle \to \langle Q'_\forall, E', \theta\rangle \tag{1}$$

and apply the "pruning" operation described later if applicable. Here ":" is cons and "$+\!\!+$" is the append for lists, and $(-)\!\downarrow_\beta$ computes the β-normal form.

Theorem 3.2. *A list of FCU unification problems E has a solution iff $\langle E, \varnothing\rangle \xrightarrow{*} \langle [], \theta\rangle$, in which θ restricted to free variables of E is a most general unifier of E.*

Implementation. In our implementation, we implement the rules in Fig. 1 with the actual transformation (1). Although this division was not taken in the original FCU [LM16], this is more handy in the functional implementation and also has been taken in the pattern unification [Nip93].

Data structure. We define the datatype of λ-terms by

```
data Term = W Id  | O Id  | C Id  | Term :@ Term | Id :.: Term | Term :#: Term | End
```

where the constructor `W` for free variables for unification, `O` for bound (and freed) variables, `:@:` for application, and `:#:` for paring with the terminal `End`. The construct `x :.: t` expresses an abstraction $\lambda x.t$. We define `Id` to be `String`.

3.2 Pruning

The pruning operation is one of the most complicated operations in FCU [LM16, Def.26]. It prunes some unification problem which can be solved immediately. Suppose an FCU unification problem $\langle Q_\forall, E, \rho\rangle$ such that E involves an equation of the form

$$X[\overline{t}] \overset{?}{=} u \trianglerighteq W[\cdots, q, \cdots]$$

where a term q is the i-th argument of m-ary free variable W, and $\overline{t}$ do not involve q. Then, the algorithm prunes this equation and obtains the substitution $W \mapsto \overline{z}.H[\overline{z}']$, where $\overline{z}'$ is obtained from $\overline{z}$ by deleting the i-th z_i, and H is fresh. This is implemented as the function

$$\texttt{prune}\ \overline{t}\ \ (\rho, u)$$

which also deals with suitable η-expansion.

```
prune tn (rho,u) = case strip (devar rho u) of
  (x:.:t',_)-> prune (O x : tn) (rho,t')
  (C _, rr) -> foldlN (prune tn) (rho,rr)
  (O x, rr) -> if O x `elem` tn
               then foldlN (prune tn) (rho,rr) else xxFail "Not unifiable"
  (W _W, sm)-> if sm `subset` tn -- all sm appear in lhs
               then rho else let vsm = mkvars sm
                                 _H  = xxNewW "H"
               in (_W, hnf (vsm, _H, eqsel vsm tn sm)) : rho
```

Here `devar` $\rho\ u$ applies a substitution ρ to a term u, `strip` splits a term as the head term and the rest of terms, and `hnf` computes the head normal form, defined in [Nip93]. `_H` is a new free variable.

3.3 Discharging operation

FCU unification requires the operation $t|^{\overline{s}}_{\overline{z}}$, which Yokoyama et al. called "discharging", that replaces terms $\overline{s}$ in t with variables $\overline{z}$. Yokoyama et al. gave a bit complicated algorithm for it [YHT04a]. Observing the similarity between the discharging operation and substitution of terms for variables, we can simply implement $t|^{\overline{s}}_{\overline{z}}$ as `discharge` θ `t` as follows.

```
discharge :: [(Term, Id)] -> Term -> Term
discharge th t = case lookup t th of
  Just y  -> O y
  Nothing -> case t of
               (x :.: t1) -> x :.: discharge th t1
               (t1 :@ t2) -> (discharge th t1) :@ (discharge th t2)
               t          -> t
```

Here, `lookup t th` looks up a term `t` in an association list `th`.

Unification function. Finally, we implement the transformation (1) and the dispatcher according to the cases in Fig. 1 as the main function `unif`. Here, `unif bvs (th,(s,t))` means to process the unification problem $\langle Q_\forall,\ (s \overset{?}{=} t),\ \theta\rangle$.

```
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)
```

For space reason, we only show the case of (flex-rigid).

```
flexrigid bvs (_F,tn,s,rho) -- s is rigid
  | occ _F rho s = xxFail "Not unifiable at flex-rigid"
  | otherwise = let zn    = mkvars  tn
                    theta = (_F, abst (zn, discharge (zip tn zn) s))
    in prune bvs tn (theta:rho, discharge (zip tn zn) s)
```

where `occ` is the occur-check function.

Other cases are also implemented in this way. The interested reader is referred to the implementation of our SOL system, which will be available from the author's homepage.

References

[Bax77] L. Baxter. *The complexity of unification*. PhD thesis, Department of Computer Science, University of Waterloo, 1977.

[Ham17] M. Hamana. How to prove your calculus is decidable: practical applications of second-order algebraic theories and computation. *Proceedings of the ACM on Programming Languages*, 1(1):22:1–22:28, 9 2017.

[KB70] D. Knuth and P. Bendix. Simple word problems in universal algebras. In *Computational Problem in abstract algebra*, pages 263–297. Pergamon Press, Oxford, 1970. included also in Automationof reasoning 2, Springer (1983), pp.342-376.

[LM16] T. Libal and D. Miller. Functions-as-Constructors Higher-Order Unification. In *Proc. of FSCD 2016*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:17, 2016.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MN98] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.*, 192(1):3–29, 1998.

[Nip93] T. Nipkow. Functional unification of higher-order patterns. In *Proc. of (LICS'93)*, pages 64–74, 1993.

[YHT04a] T. Yokoyama, Z. Hu, and M. Takeichi. Deterministic second-order patterns. *Inf. Process. Lett.*, 89(6):309–314, 2004.

[YHT04b] T. Yokoyama, Z. Hu, and M. Takeichi. Deterministic second-order patterns for program transformation. *Computer Software*, 21(5):71–76, 2004. in Japanese.