

Confluence of Pattern-Based Calculi with Finitary Matching

Sandra Alves¹, Besik Dundua^{1,3}, Mário Florido¹ and Temur Kutsia²

¹ DCC-FC & LIACC, University of Porto, Portugal

² RISC, Johannes Kepler University, Linz, Austria

³ VIAM, Ivane Javakhishvili Tbilisi State University, Georgia

Extended Abstract

Confluence of reduction means that the order how reduction is done does not affect the final result of computation. It is a very desirable semantic property for declarative programming languages, because it allows to reason about programs with respect to any convenient sequence of reductions, since the other reductions lead to the same result. In functional programming, confluence is based on the same property of λ -calculus. In logic programming, it holds due to the independence of literal selection result, called the Switching Lemma [16]. Similar theorems have been formulated in the presence of concurrency, see, e.g., [5, 8].

In this paper we present a confluence result for pattern calculi with finitary matching. Pattern calculi generalize λ -calculus by integrating pattern matching into it, abstracting on arbitrary terms, not just on variables. For instance, $\lambda(fx).(gx)$ is a well-formed expression in such a calculus, where (fx) is the pattern on which the abstraction happens.

Pattern calculi combine the functional mechanism of lambda calculus with matching capabilities of rewriting, compare β -reduction of $(\lambda(fx).(gx))(fa)$ to rewriting (fa) by the rule $(fx) \rightarrow (gx)$. Even more, as Jay states in [10], “A small typed pattern calculus supports all the main programming styles.” λ -calculus with patterns [18, 13], pure pattern calculus [11, 12], ρ -calculus [4], λ -calculus with first-order constructor patterns [17] are some examples of the integration of pattern matching and λ -calculus.

Patterns, in general, can be arbitrary terms. It is not required that all variables that appear in them to be bound by the abstraction. Those that are bound are used in matching for reduction. The free ones can be instantiated by substitutions. To distinguish these two kinds of variables, abstractions are decorated with a subscript, indicating the set of bound variables, e.g., $\lambda_{\{x\}}(fxy).(gx)$, $\lambda_{\{x,z\}}(fxy).(gx)$, $\lambda_{\{x,z\}}(fxy).(gz)$, and $\lambda_{\{x,y\}}(fxy).(gx)$.

The pattern calculi studied to far mostly consider unitary matching: Each solvable pattern matching problem has a unique solution. Despite such uniqueness, confluence is lost: For instance, if matching is done syntactically, $(\lambda_{\{x\}}(xa).x)((\lambda_{\{y\}}y.y)a)$ reduces either to $\lambda_{\{y\}}y.y$ or to $(\lambda_{\{x\}}(xa).x)a$.

To recover confluence, various restrictions have been imposed. Some restrictions concern the form of patterns, e.g., rigid pattern condition of [18]. Some others permit arbitrary patterns, but affect matching, e.g., rigid matching condition of [12]. The latter approach is general and captures various pattern calculi, such as the pure pattern calculus, λ -calculus with patterns, and ρ -calculus. Yet another generic proof of confluence is presented in [3], where the matching function is considered as a parameter and the proof is done with respect to its abstract properties. This approach can also be applied to various calculi, including λ -calculus with patterns, ρ -calculus, and a simplified version of the pure pattern calculus.

In practice, there are often the cases when matching is performed modulo some equational theory, which leads to finitely many matchers. For instance, if f is commutative, (fxy)

matches (fab) in two different ways, by the substitutions $\{x \mapsto a, y \mapsto b\}$ and $\{x \mapsto b, y \mapsto a\}$. Hence, permitting such a function in the abstraction leads to two different results: $(\lambda_{\{x,y\}}(fxy).(gx)).(fab)$ gives either (ga) or (gb) .

Extending confluence results from unitary to non-unitary matching is very useful in practice but, as it has been underlined in [3], is syntactically and semantically non-trivial and opens new challenges. This is the problem we approach here.

Computing finitely many matchers makes the resulting calculus non-deterministic, but rather than having a purely non-deterministic evaluation, we define reduction which introduces a formal sum of terms, representing all possible results of a non-deterministic computation. For instance, for commutative f , we reduce $(\lambda_{\{x,y\}}(fxy).(gx)).(fab)$ to $(ga) + (gb)$. Confluence in such a language means that regardless the order in which reduction is done, at the end of computation the result will be expressing the same alternatives (via $+$). Obviously, $+$ has to satisfy certain properties to meet this intuition: It is associative, commutative, and idempotent. Moreover, like in [7], the application distributes over $+$ both from left and from right: $(M(A+B)) = (MA) + (MB)$ and $((A+B)M) = (AM) + (BM)$. These properties are abbreviated as ACID.

Furthermore, our language permits the use of sequence variables (alongside term variables). They can be instantiated by finite, possibly empty, sequences of terms. We included them in the language because, on the one hand, sequence matching is an example of syntactic finitary matching and, on the other hand, it has interesting applications. They lead to neat, compact programs, which are easier to write: for instance, extracting duplicates from a list in one line of code. An example of such an application is the programming language of Mathematica [19] (see also [2, 15]), whose pattern matching mechanism is based on the sequence matching.

To prove confluence of our calculus, we impose restrictions on the matching function. They guarantee that no new free variable appears during reduction, and that reductions are stable both by substitution and by reduction. These properties, denoted by $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_3$, in their turn, make sure that the calculus is confluent. The matching function restrictions generalize the analogous ones, introduced for the unitary case in [3].

Our result is general, in the sense that confluence is not proved for one fixed matching function, but, rather, for an entire class of matching functions which satisfy the conditions $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_3$. Some concrete examples of matching functions that satisfy these conditions are syntactic and orderless (i.e., a generalization of commutative) sequence matching, commutative matching, and sequence matching with linear algebraic patterns.

Below we elaborate on technical details a bit more. Our terms are defined by the grammar $M, N ::= x \mid f \mid (MN) \mid (MX) \mid \lambda_V M.N \mid M + N$, where x is a term variable, X is a sequence variable, and V indicates the variables bound the abstraction. P and Q are also used for terms. As usual, application associates to the left. In the definition of bound and free variables, V plays the central role. For instance, the sets of bound and free variables of $\lambda_{\{x\}}(fx\lambda_{\{y\}}y.(xy)z).(gxu)$ are respectively $\{x, y\}$ and $\{z, u\}$.

As usual, terms are considered equal modulo α -equivalence. We work with terms in an ACID normal form. For instance, $\lambda_{\{x,z,Y\}}(fxX + gxX).(fx(gy) + fx(gY))$ is the ACID normal form of $\lambda_{\{x,z,Y\}}(fx + gx)X.fx(gy + gY + gY)$.

Substitutions are defined in the standard way, mapping term variables to terms and sequence variables to sequences. (Sequences are written within $\langle \rangle$. They are meta-expressions.) When a substitution applies to a term, it applies to all patterns occurring in the term as well, avoiding variable capture. For instance, given a substitution $\sigma = \{x \mapsto (ga), Y \mapsto \langle a, x, b \rangle\}$ and a term $M = \lambda_{\{x\}}(fxY).(XY)$, its instance under σ is $M\sigma = \lambda_{\{x'\}}(fx'axb).(x'axb)$. We write $\sigma\vartheta$ for the composition of the substitution σ with the substitution ϑ .

Our calculus has three reduction rules, parametrized by a pattern matching function *Solve*, which takes as parameters two terms P and Q and a set of bound variables V and returns a finite set of substitutions.

$$\begin{aligned} \beta_p : & (\lambda_V P.N)Q \rightarrow N\varphi_1 + \dots + N\varphi_n, \text{ where } P \text{ and } Q \text{ do not contain } + \text{ and} \\ & \text{Solve}(P \ll_V Q) = \{\varphi_1, \dots, \varphi_n\}, n \geq 1. \\ D_l : & \lambda_V (P_1 + P_2).N \rightarrow \lambda_V P_1.N + \lambda_V P_2.N. \\ D_r : & \lambda_V P.(N_1 + N_2) \rightarrow \lambda_V P.N_1 + \lambda_V P.N_2. \end{aligned}$$

Slightly abusing the notation, we use $\rightarrow$ below for a compatible closure (reduction in any context) of the union of β_p , D_l , and D_r relations. We are looking for properties of *Solve* that would guarantee confluence. For this, a technical notion of parallel reduction is needed: (It generalizes the analogous notion formulated for λ -calculus.)

$$\begin{aligned} & \frac{}{s \Rightarrow s} \quad \frac{s_1 \Rightarrow s'_1 \quad \dots \quad s_n \Rightarrow s'_n}{\langle s_1, \dots, s_n \rangle \Rightarrow \langle s'_1, \dots, s'_n \rangle} \quad \frac{M \Rightarrow M' \quad s \Rightarrow s'}{M s \Rightarrow M' s'} \\ & \frac{P \Rightarrow P' \quad N \Rightarrow N'}{\lambda_V P.N \Rightarrow \lambda_V P'.N'} \quad \frac{P_1 \Rightarrow P'_1 \quad P_2 \Rightarrow P'_2 \quad N \Rightarrow N'}{\lambda_V (P_1 + P_2).N \Rightarrow \lambda_V P'_1.N' + \lambda_V P'_2.N'} \\ & \frac{M \Rightarrow M' \quad N \Rightarrow N'}{M + N \Rightarrow M' + N'} \quad \frac{P \Rightarrow P' \quad N_1 \Rightarrow N'_1 \quad N_2 \Rightarrow N'_2}{\lambda_V P.(N_1 + N_2) \Rightarrow \lambda_V P'.N'_1 + \lambda_V P'.N'_2} \\ & \frac{P \Rightarrow P' \quad N \Rightarrow N' \quad Q \Rightarrow Q'}{(\lambda_V P.N)Q \Rightarrow N'\varphi_1 + \dots + N'\varphi_n} \end{aligned}$$

In the last rule P' and Q' do not contain $+$ and $\text{Solve}(P' \ll_V Q') = \{\varphi_1, \dots, \varphi_n\}$, $n \geq 1$.

Based on this definition, the conditions imposed on *Solve* are formulated as follows:

- C₁**: No new free variables should be introduced: If $\varphi \in \text{Solve}(P \ll_V Q)$, then $\text{Dom}(\varphi) = V$ and $\text{fv}(\text{Ran}(\varphi)) \subseteq \text{fv}(Q) \cup (\text{fv}(P) \setminus V)$.
- C₂**: *Solve* should be stable under substitutions: If $\text{Solve}(P \ll_V Q) = \{\varphi_1, \dots, \varphi_n\}$, $n \geq 1$, then for all ϑ with $\text{Var}(\vartheta) \cap V = \emptyset$, we have $\text{Solve}(P\vartheta \ll_V Q\vartheta) = \{(\varphi_1\vartheta)|_V, \dots, (\varphi_n\vartheta)|_V\}$.¹
- C₃**: *Solve* should be stable under reduction: If $\text{Solve}(P \ll_V Q) = \{\varphi_1, \dots, \varphi_n\}$, $n \geq 1$, $P \Rightarrow P'$, and $Q \Rightarrow Q'$, then $\text{Solve}(P' \ll_V Q') = \{\varphi'_1, \dots, \varphi'_m\}$, $m \geq 1$, and
 - (a) for all $1 \leq i \leq n$ there exists $1 \leq j \leq m$ such that $\varphi_i \Rightarrow \varphi'_j$ and
 - (b) for all $1 \leq j \leq m$ there exists $1 \leq i \leq n$ such that $\varphi_i \Rightarrow \varphi'_j$.

These conditions are sufficient for confluence of our calculus. The corresponding theorem, which is the main result of this work, can be proved in the style of Tait and Martin-Löf [1]: Reflexive-transitive closures of $\rightarrow$ and $\Rightarrow$ are the same and $\Rightarrow$ has the diamond property. Therefore, the reflexive-transitive closure of $\Rightarrow$ is confluent. Hence, the reflexive-transitive closure of $\rightarrow$ is confluent, which is what we want.

An example of *Solve* which satisfies **C₁**, **C₂**, **C₃** is sequence matching defined as follows: First, a matching equation between P and Q is denoted by $P \ll_V^? Q$. Its solution (a matcher) is a substitution φ such that $P\varphi \doteq Q$, where $\doteq$ stands for equality modulo α and the given equational theory. A matching problem Π is a finite set of matching equations. A state is a

¹ $(\varphi_n\vartheta)|_V$ stands for the restriction of the substitution $\varphi_n\vartheta$ on V .

pair $\langle \Pi, \vartheta \rangle$ of a matching problem and a substitution. Rules for matching equations transform an equation into a set of equations: $P \ll_V^? Q \rightsquigarrow_\varphi \Pi'$, where φ is the substitution computed at the transformation step. State transformation by a rule $P \ll_V^? Q \rightsquigarrow_\varphi \Pi'$ is defined as $\langle \{P \ll_V^? Q\} \cup \Pi, \vartheta \rangle \rightsquigarrow \langle (\Pi \cup \Pi')\varphi, \vartheta\varphi \rangle$. There are four rules, for deletion, decomposition, term and sequence variable elimination, adapted from [14]: (ε stands for the identity substitution. Rules are applied modulo α -equivalence.)

Del : $M \ll_V^? M \rightsquigarrow_\varepsilon \emptyset$, where $\text{fv}(M) \cap V = \emptyset$.

Dec : $M_1 M_2 \ll_V^? N_1 N_2 \rightsquigarrow_\varepsilon \{M_1 \ll_V^? N_1, M_2 \ll_V^? N_2\}$,

TVE : $x \ll_V^? N \rightsquigarrow_{\{x \mapsto N\}} \emptyset$, where $x \in V$.

SVE : $M X \ll_V^? N t_1 \cdots t_n \rightsquigarrow_{\{X \mapsto \langle t_1, \dots, t_n \rangle\}} \{M \ll^? N\}$, where $X \in V$ and $n \geq 0$.

The matching function $\text{Solve}(P \ll_V Q)$ is defined as a partial function from P, Q, V to a set of substitutions under the following conditions:

Cond.1. If $P = x$ and $V = \{x\}$, then $\text{Solve}(P \ll_V Q) = \{\{x \mapsto Q\}\}$.

Cond.2. If $P \neq x$, assume the following conditions are satisfied:

Cond.2.1: $V \subseteq \text{fv}(P)$.

Cond.2.2: P and Q are redex-free.

Cond.2.3: Q does not contain free variables.

Then we transform $P \ll_V^? Q$ by the rules above in all possible ways as long as possible. Substitutions φ from the success states $\langle \emptyset, \varphi \rangle$ are collected in the set of matchers.

Cond.3. Otherwise, $\text{Solve}(P \ll_V Q)$ is undefined.

It can be proved that Solve defined in this way satisfies **C**₁, **C**₂, and **C**₃. The conditions Cond.2.2 (Q is redex-free) and Cond.2.3 mean that Q is a rigid value. This matching imposes call-by-rigid-value strategy. One can define other matching functions that relax call-by-value strategy, but impose more restrictions on the patterns (e.g., so called linear algebraic patterns).

Note that if we remove Cond.2.3, confluence is lost: $M = (\lambda_{\{Z\}}(f Z).(\lambda_{\{X,Y\}}(f X Y).(f X))(f Z))(f a b)$ reduces to two different normal forms: $f + f a b$ and $f + f a + f a b$. In this case **C**₂ is violated: Take $P = (f X Y)$, $Q = (f Z)$, and $\vartheta = \{Z \mapsto \langle a, b \rangle\}$. Then $\text{Solve}(P \ll_{\{X,Y\}} Q) = \{\{X \mapsto \epsilon, Y \mapsto Z\}, \{X \mapsto Z, Y \mapsto \epsilon\}\}$ and $\text{Solve}(P \ll_{\{X,Y\}} Q\vartheta) = \{\{X \mapsto \epsilon, Y \mapsto \langle a, b \rangle\}, \{X \mapsto a, Y \mapsto b\}, \{X \mapsto \langle a, b \rangle, Y \mapsto \epsilon\}\}$.

It might be tempting to leave out V from $\lambda_V P.N$ and assume that the abstraction binds all variables in the pattern. However, it causes a serious problem: Bound variables can get freed, which might lead to the loss of confluence: Consider $\lambda(\lambda x.c)y.y \rightarrow \lambda c.y$ and $\lambda(\lambda x.c)y.y =_\alpha \lambda(\lambda x.c)z.z \rightarrow \lambda c.z$. Note that in this case **C**₁ is violated. In comparison: When V is explicit, then, on the one hand, $\lambda_{\{y\}}(\lambda_{\{x\}}x.c)y.y \rightarrow \lambda_{\{y\}}c.y$, and on the other hand $\lambda_{\{y\}}(\lambda_{\{x\}}x.c)y.y =_\alpha \lambda_{\{z\}}(\lambda_{\{x\}}x.c)z.z \rightarrow \lambda_{\{z\}}c.z$. Hence, the obtained terms $\lambda_{\{y\}}c.y$ and $\lambda_{\{z\}}c.z$ are α -equivalent.

We remark that in case of non-confluence by “redex breaking”, **C**₃ is violated. For instance, $(\lambda_{\{x,y\}}(xy).(yx))(\lambda_{\{z\}}z.z)(f a)$ reduces to both $(f a)\lambda_{\{z\}}z.z$ and $(a f)$, breaking the redex $(\lambda_{\{z\}}z.z)(f a)$. Violation of **C**₃ can be seen, if we take $P = (xy)$, $Q = (\lambda_{\{z\}}z.z)(f a)$ and get $P \rightrightarrows P' = P$, $Q \rightrightarrows Q' = (f a)$, $\text{Solve}(P \ll_{\{x,y\}} Q) = \{\{x \mapsto \lambda_{\{z\}}z.z, y \mapsto (f a)\}\}$, $\text{Solve}(P' \ll_{\{x,y\}} Q') = \{\{x \mapsto f, y \mapsto a\}\}$.

Specific pattern calculi with finitary matching are obtained from the described calculus by adding rewrite rules. The proof of confluence of such a specific calculus uses Yokouchi-Hikita’s lemma [20, 6] and is analogous to the proof of the same result for unitary matching [9].

Acknowledgments

This research has been partially supported by the FCT fellowship (ref. SFRH/BD/62058/2009), by the Austrian Science Fund (FWF) under the project SToUT (P 24087-N18), and the Rustaveli Science Foundation under the grants DI/16/4-120/11 and FR/611/4-102/12.

References

- [1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984. Revised edition.
- [2] B. Buchberger. Mathematica as a rewrite language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, FLOPS 1996*, pages 1–13. World Scientific, 1996.
- [3] H. Cirstea and G. Faure. Confluence of pattern-based calculi. In F. Baader, editor, *RTA*, volume 4533 of *LNCS*, pages 78–92. Springer, 2007.
- [4] H. Cirstea and C. Kirchner. The rewriting calculus - parts I and II. *Logic Journal of the IGPL*, 9(3), 2001.
- [5] M. Codish, M. Falaschi, K. Marriott, and W. H. Winsborough. A confluent semantic basis for the analysis of concurrent constraint logic programs. *J. Log. Program.*, 30(1):53–81, 1997.
- [6] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *J. ACM*, 43(2):362–397, 1996.
- [7] A. Díaz-Caro and B. Petit. Linearity in the non-deterministic call-by-value setting. In C.-H. L. Ong and R. J. G. B. de Queiroz, editors, *WoLLIC*, volume 7456 of *LNCS*, pages 216–231. Springer, 2012.
- [8] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theor. Comput. Sci.*, 183(2):281–315, 1997.
- [9] G. Faure. *Structures et modèles de calculs de réécriture*. PhD thesis, Université Henri Poincaré, Nancy, France, 2007.
- [10] B. Jay. *Pattern Calculus*. Springer, 2009.
- [11] C. B. Jay and D. Kesner. Pure pattern calculus. In P. Sestoft, editor, *ESOP*, volume 3924 of *LNCS*, pages 100–114. Springer, 2006.
- [12] C. B. Jay and D. Kesner. First-class patterns. *J. Funct. Program.*, 19(2):191–225, 2009.
- [13] J. W. Klop, V. van Oostrom, and R. C. de Vrijer. Lambda calculus with patterns. *Theor. Comput. Sci.*, 398(1-3):16–31, 2008.
- [14] T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007.
- [15] T. Kutsia. Flat matching. *J. Symb. Comput.*, 43(12):858–873, 2008.
- [16] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.
- [17] S. L. Peyton Jones and P. Wadler. *The Implementation of Functional Programming Languages*, chapter 4: Structured Types and the Semantics of Pattern Matching. Prentice Hall, 1987.
- [18] V. van Oostrom. Lambda calculus with patterns. Technical Report IR-228, Vrije Universiteit, Amsterdam, 1990.
- [19] S. Wolfram. *The Mathematica book*. Wolfram-Media, fifth edition, 2003.
- [20] H. Yokouchi and T. Hikita. A rewriting system for categorical combinators with multiple arguments. *SIAM J. Comput.*, 19(1):78–97, 1990.