

The dynamic pattern calculus as a higher-order pattern rewriting system

Vincent van Oostrom

Philosophy
Utrecht University
Utrecht, The Netherlands
v.vanoostrom@uu.nl

Femke van Raamsdonk

Department of Computer Science
VU University
Amsterdam, The Netherlands
f.van.raamsdonk@vu.nl

We show that Jay and Kesner’s dynamic pattern calculus can be embedded into a higher-order pattern rewriting systems in the sense of Nipkow. Metatheoretical results, such as confluence and standardisation, are shown to hold for the dynamic pattern calculus as a consequence of this embedding. The embedding also opens a way to study the semantics of Jay’s programming language *bondi* based on pattern matching.

Introduction Pattern calculi are extensions of the λ -calculus where the usual abstraction over a variable, λx , is replaced by a more general abstraction over a *pattern*, λP . The intuition is that $\lambda P.M$ is a function that requires its argument to be of the shape P , or, the application $(\lambda P.M) N$ can be reduced only if the argument N matches the pattern P . For example, the application $(\lambda(x,y).x) N$ can be reduced only if $N = (N_1, N_2)$, and in that case it reduces to N_1 . When this general idea is worked out to obtain a decent λ -calculus with patterns, various questions arise. What is a good pattern? What is matching?

The motivation for studying lambda calculi with patterns originates from the work of Peyton Jones on the implementation of functional programming languages [9]. After that, various lambda calculi with patterns have been introduced [8, 5, 3, 2]. In such studies, a main concern is how to define the calculus in order to have λ -like properties, such as uniqueness of normal forms, confluence, finiteness of developments, and normalization properties. In order to define the calculus, and to study such properties, we need the usual notions of binding, α -equivalence, redexes, orthogonality, and machinery such as residuals, descendants, projections.

Because the definition of a pattern calculus depends on various design choices, there are many different pattern calculi. A priori, for every such calculus we need to define the notions and machinery as mentioned above, that is, if we want to prove for instance confluence, or the normalization of a strategy. More generally, also when studying other extensions of the lambda calculus, one often adapts notions and machinery of the lambda calculus to the actual calculus.

This is the reason that higher-order rewriting was introduced: to provide a general setting for obtaining uniform results for term rewriting systems with binders. Results available for higher-order rewriting concern for instance confluence and termination [11, 4], standardization [1] and normalisation of several strategies. Unfortunately, higher-order rewriting is used not as often as it could be used. There are various reasons for this; we identify two of them. First, it is often not trivial to see a particular rewriting system with binders as an instance of the general format of higher-order rewriting. Second, there is not just one notion of higher-order rewriting, but various such general formats have been introduced which means that the user has to make a choice.

In the present work we are interested in the dynamic pattern calculus (dpc) introduced by Jay and Kesner[3]. We propose an embedding of dpc into an higher-order rewrite system. This is in the spirit

of [5, Remark 8]. We work with higher-order pattern rewrite systems (HRSs) as defined by Nipkow [7, 6]; this choice is convenient but not essential. We represent dpc as a HRS, following step by step the definitions of terms, substitution, matching, and rewriting. Then we show how this embedding permits one to apply general higher-order rewriting results to this HRS, to obtain properties of dpc.¹

Statics of the dpc We recapitulate the terms and substitutions of the (strict) dynamic pattern calculus (dpc), following the account given in [2, Section 5.2] as closely as possible. The main feature is the *case*, which intuitively is a function that in order to be evaluated requires its argument to be of a form specified in the *pattern*. The *term syntax* of the dpc is inductively defined by the following grammar for *variable*, *matchable*, *application*, and *case*:

$$t ::= x \mid \hat{x} \mid tt \mid [\theta]t \rightarrow t$$

with θ a finite list of pairwise distinct variables. As the notation should suggest to every variable x corresponds a matchable $\hat{x}$ and this correspondence is a bijection. The grammar for *contexts* C is the one for terms extended with an option for $\square$ (a *hole*). Application is assumed to be left-associative, and case right-associative. The p in a case $[\theta]p \rightarrow t$ is said to be its *pattern*. A main novel feature of dpc compared to other pattern calculi is the explicit list θ of pattern variables. Two examples of expressions of the term syntax are $[x, y]\hat{z}\hat{x}\hat{y} \rightarrow x$ and $[x]\hat{x} \rightarrow [y]x\hat{y} \rightarrow y$. Both variables and matchables occur either free or bound: in a case expression $[\theta]p \rightarrow t$, a variable x in θ binds the occurrences of x in t , and also the occurrences of the corresponding matchable $\hat{x}$ in p , but only those: e.g. in $[x]x\hat{x} \rightarrow x\hat{x}$, the occurrence of x in the pattern is free and the other one is bound, whereas for $\hat{x}$ the reverse is the case.

A *substitution* is a partial function from variables to expressions of the term syntax. A substitution σ is lifted to a *matchable* substitution $\hat{\sigma}$ and a *variable* substitution, also written σ , informally as follows. The matchable substitution $\hat{\sigma}$ applied to a matchable $\hat{x}$ yields $\sigma(x)$ if σ is defined on the corresponding variable x of $\hat{x}$; the matchable substitution is the identity on the other matchables and on all variables, and is compatible with the term syntax. The variable substitution σ yields $\sigma(x)$ if σ is defined on the variable x ; the variable substitution is the identity on the other variables and on all matchables, and is compatible with the term syntax. For a case expression we assume that the substitution σ avoids the variables of θ in the sense that it is not defined on variables in θ , and moreover σ does not introduce variables nor matchables occurring in θ .

Terms of the dpc are defined as α -equivalence classes of the term syntax. Contexts are not taken up to α -equivalence. For instance, $[x]x\hat{x} \rightarrow x\hat{x}$ and $[y]x\hat{y} \rightarrow y\hat{x}$ are distinct as expressions of the term syntax, but denote the same term, since the expressions are α -equivalent.

Embedding the dpc statics We show that the terms of the dpc can be faithfully embedded into a subset of the terms over a (well-chosen) higher-order signature, in such a way that both matchable and variable substitution are represented by higher-order substitution. That is achieved by employing two base types: $\mathbb{M}$ for matchables and $\mathbb{T}$ for terms. For convenience we assume two disjoint infinite sets of variables $\mathcal{M} = \{x, y, \dots\}$ of type $\mathbb{M}$ and $\mathcal{T} = \{X, Y, \dots\}$ of type $\mathbb{T}$, and bijective mappings between both: $\uparrow$ maps a variable x of type $\mathbb{M}$ to the variable X of type $\mathbb{T}$, and $\downarrow$ maps a variable X of type $\mathbb{T}$ to the variable x of

¹The notion of pattern of the dynamic pattern calculus and that of higher-order pattern rewrite systems are related, as shown by our embedding, but not to be confused! Roughly speaking, patterns in the sense of the former *giving rise to a matching that is defined* will be related to *substitution instances* of patterns in the sense of the latter.

type $\mathbb{M}$. We take the following (infinite) signature Π :

$$\begin{aligned} * & : \mathbb{M} \rightarrow \mathbb{T} \\ @ & : \mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T} \\ \text{case}_n & : (\mathbb{M}^n \rightarrow \mathbb{T}) \rightarrow (\mathbb{T}^n \rightarrow \mathbb{T}) \rightarrow \mathbb{T}, \text{ for all natural numbers } n \end{aligned}$$

use a hole $\square : \mathbb{T}$ in order to embed contexts, and define the following embedding of the term syntax of the dpc into the higher-order preterms (so not, yet, modulo $\alpha\beta\eta$) over Π :

t	$(t)_{ho}$
x	X
$\hat{x}$	x^*
$t_1 t_2$	$@((t_1)_{ho}, (t_2)_{ho})$
$[\vec{x}] t_1 \rightarrow t_2$	$\text{case}(\vec{x}.(t_1)_{ho}, \vec{X}.(t_2)_{ho})$

where we use the following notations: x^* abbreviates $*(x)$, and, leaving n implicit, $\text{case}(\vec{x}.T, \vec{X}.S)$ abbreviates $\text{case}_n(x_1 \dots x_n.T, X_1 \dots X_n.S)$. Note the finite list of pairwise distinct variables $\vec{x}$ in a case expression of the dpc is split into two lists in its higher-order embedding, a list $\vec{x}$ of variables for matchables in the first argument of the case, and a list $\vec{X}$ of variables for terms its second argument. For example, the embedding of $[x, y] \widehat{x} \widehat{y} \rightarrow x$ is $\text{case}(xy.@(@(\vec{z}^*, x^*), y^*), XY.X)$, the embedding of $[x] \widehat{x} \rightarrow [y] x \widehat{y} \rightarrow y$ is $\text{case}(x.x^*, X.\text{case}(y.@(X, y^*), Y.Y))$, and the embedding of $[x] x \widehat{x} \rightarrow x \widehat{x}$ is $\text{case}(x.@(X, x^*), X.@(X, x^*))$.

The notions of freeness for matchables and variables, of substitution, and of α -equivalence all correspond between dpc and its embedding. For example, in the higher-order term $\text{case}(x.@(X, x^*), X.@(X, x^*))$ above, the occurrence of X (of type $\mathbb{T}$; the embedding of x) in the first argument of the case is free and the one in the second argument is bound, whereas for x (of type $\mathbb{M}$; the embedding of $\widehat{x}$) the reverse is the case, corresponding to how the variable x and the matchable $\widehat{x}$ are bound in the dpc case $[x] x \widehat{x} \rightarrow x \widehat{x}$. That the embedding respects α -equivalence is illustrated by the expression $[y] x \widehat{y} \rightarrow y \widehat{x}$ (α -equivalent to the above expression of the dpc term syntax) the embedding of which yields $\text{case}(y.@(X, y^*), Y.@(Y, x^*))$, (which is indeed α -equivalent to the above higher-order preterm).

Theorem 1 *The map $(\)_{ho}$ is a bijection between dpc and DPC.*

Here *dpc* denotes the set of terms of the dpc and *DPC* denotes the set of higher-order terms over Π (of type $\mathbb{T}$ and with free variables of base type). The bijection extends naturally to contexts.

Dynamics of the dpc We recapitulate from [3] the rewrite relation of the dynamic pattern calculus and the notions of matching on which it is based (see that paper for details). In the dpc, $\{t // [\theta] p\}$ is employed to denote *basic* matching of a term p against a term t for a list of pairwise distinct variables θ binding in p ,² which is the partial function on these ingredients defined by the following clauses, in order:

$$\begin{aligned} \{t // [\theta] \widehat{x}\} &= \text{some } \{t/x\} && \text{if } x \in \theta \\ \{\widehat{x} // [\theta] \widehat{x}\} &= \text{some } \{\} && \text{if } x \notin \theta \\ \{t_1 t_2 // [\theta] p_1 p_2\} &= \{t_1 // [\theta] p_1\} \uplus \{t_2 // [\theta] p_2\} && \text{if } p_1 p_2, t_1 t_2 \text{ matchable forms} \\ \{t // [\theta] p\} &= \text{none} && \text{otherwise, if } p, t \text{ matchable forms} \end{aligned}$$

²Applying the variable convention, we will assume that the free variables and matchables in t do not occur in θ .

where matchable forms are the subset of the terms defined by the grammar:

$$\begin{array}{lll} d & ::= & \hat{x} \mid dt \quad (\text{data structure}) \\ m & ::= & d \mid [\theta]t \rightarrow t \quad (\text{matchable form}) \end{array}$$

Note that $t_1 t_2$ is a matchable form if and only if t_1 is a data structure.

If one of the first three basic matching clauses applies, then it *succeeds* yielding some substitution. If none of these clauses applies but the fourth clause does, then basic matching *fails*. If also the fourth clause does not apply then basic matching is *undefined*. In the third clause, the disjoint union $\uplus$ of two basic matchings is undefined if one of them is. Otherwise it fails if at least one of the two basic matchings does, or if both succeed but have non-disjoint domains. *Matching*, denoted by $\{t / [\theta] p\}$ is defined as basic matching $\{t // [\theta] p\}$, except when the latter succeeds in a substitution with a domain distinct from θ in which case it fails yielding none. *Reduction* is driven by the β_m -rule $([\theta] p \rightarrow s) t \rightarrow \{t / [\theta] p\} s$ where its right-hand side must be a term resulting from the application of a defined match $\{t / [\theta] p\}$ to s .

An example of (basic) matching: $\{\hat{z}\hat{u}\hat{v} // [x, y] \hat{z}\hat{x}\hat{y}\} = \text{some } \{\} \uplus \text{some } \{\hat{u}/x\} \uplus \text{some } \{\hat{v}/y\}$. Note that $\{\hat{w} // [x, y] \hat{z}\} = \text{none}$, and hence the matching $\{\hat{w}\hat{u}\hat{v} // [x, y] \hat{z}\hat{x}\hat{y}\}$ fails.

A basic matching is undefined intuitively if it originates from an application of a case to an argument where either the pattern of the case, or the argument is not a matchable form, or neither the pattern nor the argument is not a matchable form. Typical examples of terms that are no matchable forms: y , yt with t some *dpc* term, and $([y]\hat{y} \rightarrow \hat{y})t$ with t some *dpc* term. For example, the following terms do not reduce, because the induced basic matching is undefined: $([x]y \rightarrow x)\hat{u}$, and $([x]\hat{y} \rightarrow x)u$, and $([x](y\hat{x}) \rightarrow x)\hat{u}$.

Let Ω be a non-terminating term. In the present strict setting, where the disjoint union of two matchings is undefined if at least one of them is, the term $([\hat{z}\hat{x}\hat{x} \rightarrow \hat{x}](\hat{z}\hat{u}\Omega)$ does not perform a step, because the basic matching part $\{\Omega // [\hat{x}]\}$ is undefined, because Ω is not a matchable form (it never will be one). For non-strict matching as in [3], where the disjoint union of two matchings is none if at least one of them is (allowing the other to be undefined), $([\hat{z}\hat{x}\hat{x} \rightarrow \hat{x}](\hat{z}\hat{u}\Omega)$ does reduce; it reduces with a fail step to $([x]\hat{x} \rightarrow x)$, because of the following basic matching part: $\{\hat{u} // [\hat{x}]\} = \text{none}$.

An example of a rewrite step in the dpc: We have the step $([x, y] \hat{z}\hat{x}\hat{y} \rightarrow x)(\hat{z}\hat{u}\hat{v}) \rightarrow_{\beta_m} \hat{u}$. Note that $([x, y] \hat{z}\hat{x}\hat{y} \rightarrow x)\hat{w}\hat{u}\hat{v}$ performs a none step to $([x]\hat{x} \rightarrow x)$.

Embedding of dpc dynamics We now define an embedding of the reduction relation of the dpc into the rewrite relation induced by a set of higher-order pattern rewrite rules over Π .³ The rewrite rules are of the following two shapes (with the same left-hand side):

$$\begin{array}{ll} @((\text{case}(\theta.P, \Theta.Z(\Theta))), T) & \rightarrow Z(\vec{S}) \quad (\beta_{\top}) \\ & \rightarrow \text{case}(x.x^*, X.X) \quad (\beta_{\perp}) \end{array}$$

Given specific higher-order terms P , T and $\vec{S}$, the left-hand side uniquely determines its right-hand side, among the two possibilities (top and bottom) presented. Omitting many details, the idea is that P and T are taken from a set of higher-order patterns the matching of which captures, exactly, the matching procedure of dpc. Although the exact specification of these is technical (it has to be as it reflects the matching procedure) the idea is simple: P and T are terms restricted to share the same applicative context (context consisting of applications only) the holes of which are then filled by ‘shallow’ terms (variables, matchables, and applications and cases having (higher-order) variables as arguments):

³One could simply define the steps on the higher-order terms *DPC* by transferring the steps on the terms *dpc* of the dpc along the bijection between both. However, that would defeat the goal of this work, as it would not allow for transferring higher-order rewriting meta-theory back to the dynamic pattern calculus.

- If then there's a match between the ways the contexts for both P and T are filled, this gives rise to an appropriate term-vector $\vec{S}$ and a *succeed* rewrite rule of the top shape. An application of the rule leads to an appropriate instantiation of (the matchables and variables in) T which is then used to instantiate the vector $\vec{S}$ that is passed to the body in the right-hand side of the rule.
- However, if there's a mismatch between the fillings of the applicative context shared by P and T , this induces a *fail* rewrite rule of the bottom shape. Such a mismatch indicates that there's no way that an instance of T can be further reduced to a term matched by P .

Although the pairs P, T constitute an infinite set, they can be generated by a finite context-free grammar, making the calculus amenable to direct implementation in CRSX (or rather an extension of it allowing matching specified by such grammars).

For example, setting $\theta = x$, $P = x$ and $T = Z$, we see that only an instance of the first line is applied, giving rise to the succeed rule $@((\text{case}(x.x^*, X.Z(X))), W) \rightarrow Z(W)$ which is the pattern calculi version of the usual higher-order term rewriting rendering of the ordinary β -rule of the λ -calculus (in this case the pattern $x.x$ does not put any restrictions on the allowed shape of the argument W , and is therefore simply not present in such renderings). The dpc-step $([x, y] \widehat{z\hat{x}\hat{y}} \rightarrow x) (\widehat{z\hat{u}\hat{v}}) \rightarrow_{\beta_m} \widehat{u}$. considered above, is embedded as follows: $@(\text{case}(xy. @(@ (z^*, x^*), y^*), XY.X), @(@ (z^*, u^*), v^*)) \rightarrow u^*$ using $P = \lambda xy. @(@ (z^*, x), y)$, and $Z = \lambda xy. xm$ and $\vec{Z} = u^*, v^*$.

Theorem 2 *The rules β_+ and β_- together form a left-linear pattern higher-order rewrite system B_m over Π . The map $(\)_{ho}$ is a bijection between the rewrite relation $\rightarrow_{\beta_m}$ of the dpc, and the restriction to DPC of the higher-order rewrite relation.*

Formally, the higher-order rules have to be seen as conditional rules in that their instantiation has to respect the (in)equality of the matchable variables. By inspection/design B_m is even seen to be (conditionally) orthogonal, so we have:

Theorem 3 *The orthogonal conditional HRS B_m is confluent on DPC; has finite (family) developments; enjoys standardisation; has that the needed (necessary) strategy is normalising.*

Although this result is already powerful, much more comes for free from the embedding, e.g. confluence results when modularly combining the dpc with other rewrite system, termination techniques for higher-order rewriting, etc. Apart from this the embedding also should facilitate the formal development (Coq) of (results for) the dpc and related pattern calculi, by allowing reuse of extant formalisations. Although the set of higher-order rewrite rules of our embedding is infinite, it is very regular, as our presentation above already suggests, and can in fact be described by a finite grammar (parsing is easy). We expect that is sufficient to make the dpc, and programming languages such as bondi based upon it, amenable to implementation by means of extant higher-order rewriting tools such as CRSX [10], after extending such with a facility to deal with rules generated by a grammar.

Acknowledgements We thank Eduardo Bonelli, Delia Kesner, Jan Willem Klop, Carlos Lombardi, Alejandro Ríos, Roel de Vrijer and the anonymous referees of an earlier submission for discussions and constructive remarks.

References

- [1] H.J. Sander Bruggink (2008): *Equivalence of reductions in higher-order rewriting*. Ph.D. thesis, Utrecht University.
- [2] B. Jay (2009): *Pattern Calculus, Computing with Functions and Structures*. Springer.
- [3] B. Jay & D. Kesner (2009): *First-class patterns*. *Journal of Functional Programming* 19(2), pp. 191–225.

- [4] J.-P. Jouannaud & A. Rubio (2007): *Polymorphic higher-order recursive path orderings*. *Journal of the ACM* 54(1), pp. 1–48.
- [5] J.W. Klop, V. van Oostrom & R. de Vrijer (2008): *Lambda calculus with patterns*. *TCS* 398, pp. 16–31.
- [6] R. Mayr & T. Nipkow (1998): *Higher-Order Rewrite Systems and their Confluence*. *TCS* 192, pp. 3–29.
- [7] Tobias Nipkow (1991): *Higher-order critical pairs*. In: *Proceedings of LICS'91*, IEEE Computer Society Press, pp. 342–349.
- [8] V. van Oostrom (1990): *Lambda Calculus with Patterns*. Technical Report IR 228, Vrije Universiteit, Amsterdam. Digital object available from author's homepage.
- [9] S.L. Peyton Jones (1987): *The implementation of functional programming languages*. Prentice Hall.
- [10] K.H. Rose (2011): *CRSX – Combinatory Reduction Systems with Extensions*. In Manfred Schmidt-Schauß, editor: *RTA 2011, LIPIcs* 10, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 81–90.
- [11] Terese (2003): *Term Rewriting Systems*. *Cambridge Tracts in Theoretical Computer Science* 55, CUP.