# Deterministic Second-order Patterns

Tetsuo Yokoyama [a], Zhenjiang Hu [a,b], Masato Takeichi [a]

[a] *Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, JAPAN*

[b] *PRESTO 21, Japan Science and Technology Agency*

**Abstract**

Second-order patterns, together with second-order matching, enable concise specification of program transformation, and have been implemented in several program transformation systems. However, second-order matching in general is nondeterministic, and the matching algorithm is so expensive that the matching is NP-complete. It is orthodox to impose constraints on the form of higher-order patterns so as to obtain the desirable matches satisfying certain properties such as decidability and finiteness. In the context of unification, Miller's *higher-order patterns* have a single most general unifier. In this paper, we relax the restriction of his patterns without changing determinism in the context of matching instead of unification. As a consequence, our *deterministic second-order patterns* cover a wide class of useful patterns for program transformation. The time complexity of our deterministic matching algorithm is linear in the size of a term for a fixed pattern.

*Key words:* Second-order pattern matching, Functional programming, Program derivation, Program transformation, Fusion transformation

*1991 MSC:* 68W99

## 1. Introduction

Second-order patterns, together with second-order matching, enable concise specification of program transformation, and have been implemented in several program transformation systems [4,10]. However, second-order matching in general is nondeterministic [9] (there is more than a single match). It is orthodox to restrict the form of higher-order patterns to generate the desirable matches satisfying certain properties such as decidability [12] and finiteness [6].

In the context of unification, Miller defined a certain class of *higher-order patterns* [11] that are deterministic, i.e., patterns have at most a single most general unifier. He required that free variables should appear as the head of a term where the arguments are distinct bound variables. For example, the pattern $\lambda x\, y.\, p\; y\; x$ is valid, since the arguments of the free variable $p$ are distinct bound variables $y$ and $x$. Miller's higher-order patterns, however, are too restrictive for program transformations.

In this paper, we relax the restriction of Miller's patterns by allowing the arguments to be terms, so that our *deterministic second-order patterns* cover

*Email addresses:* `yokoyama@ipl.t.u-tokyo.ac.jp` (Tetsuo Yokoyama), `hu@mist.i.u-tokyo.ac.jp` (Zhenjiang Hu), `takeichi@mist.i.u-tokyo.ac.jp` (Masato Takeichi).

a wide class of useful patterns for program transformations. Consider, for example, the following fusion transformation rule, which eliminates unnecessary intermediate data structures, in Haskell-like notation [2]:

$$\frac{\forall x, y.\, x \otimes f\ y = f(x \oplus y)}{f\ .\ \mathit{foldr}\ (\oplus)\ e = \mathit{foldr}\ (\otimes)\ (f\ e)}$$

which says that a composition of function $f$ with a *foldr* can be fused into a single *foldr*, provided that one can find a function $\otimes$ satisfying the side condition, namely $x \otimes f\ y = f\ (x \oplus y)$. The key step of discovering a suitable $\otimes$ is actually a higher-order matching problem. Consider fusing *sum* and *foldr* $(\lambda x\, y.\ x * x : y)$ $[\,]$. To see this, expanding the right-hand side of the fusion condition, we get:

$$\begin{aligned}&\lambda x\, y.\, f\ (x \oplus y)\\ =\ &\lambda x\, y.\, \mathit{sum}\ (x * x : y)\\ =\ &\lambda x\, y.\, x * x + \mathit{sum}\ y\end{aligned}$$

We then obtain $\otimes$ by matching the resulting term, $\lambda x\, y.\, x{*}x{+}\mathit{sum}\, y$, with the pattern $\lambda x\, y.\, x{\otimes}\mathit{sum}\, y$. This pattern is beyond Miller's higher-order pattern, and the match $\{\otimes \mapsto \lambda y_1\, y_2.\, y_1 * y_1 + y_2\}$ cannot be obtained by first-order matching. On the other hand, our approach can deal with such patterns and guarantee a unique match.

## 2. Deterministic Second-order Patterns

We consider simply-typed lambda *terms*. Terms are built recursively from constants, variables, $\lambda$-abstractions, and function applications.

$$T = c \mid v \mid \lambda x. T \mid T\ T$$

Given two terms $T_1$ and $T_2$, we write $T_1 \trianglelefteq T_2$ if $T_1 =_\alpha T_2$ or $T_1$ is a proper subterm of $T_2$, up to $\alpha$-equivalence. For a term $v\ T_1\ \cdots\ T_n$, we call $v$ the *head* of the term and $T_1, \ldots, T_n$ the *arguments* of $v$. A term $T$ is called $\eta$-(short) normal if $T$ has no $\eta$-redex.

Let $FV$ be the function mapping from a term to the set of its free variables. We call the term $T$ *closed* if $FV(T) = \{\,\}$. For readability we sometimes use infix notation, so $x + y$ denotes the term $(+)\ x\ y$.

A substitution (or *match*) is a partial function from variables to closed terms like $\phi = \{p \mapsto \lambda x.\ x\ b\}$. The domain of substitution $\phi$ is written as $\mathit{dom}(\phi)$. Given substitutions $\phi$ and $\psi$, the composition of substitutions is written as $\phi\,.\,\psi$. The *quasi-composition* of substitutions $\phi \circ \psi$ is defined as $\phi\,.\,\psi$ if the same variables in domains have the same ranges:

$$\forall v \in \mathit{dom}(\phi) \cap \mathit{dom}(\psi)\,.\, \phi\ v =_{\alpha\beta\eta} \psi\ v$$

Where the equality operator $(=_{\alpha\beta\eta})$ is modulo under $\alpha\beta\eta$-conversion. Otherwise, $\phi \circ \psi$ is *fail*. We use a special match *fail* that is the zero of match composition, i.e., $\mathit{fail} \circ m = m \circ \mathit{fail} = \mathit{fail}$.

Let $T_0$ be the set of base types. The set of types $T$ is defined as follows.

$$\alpha \in T_0 \Rightarrow \alpha \in T, \quad \alpha, \beta \in T \Rightarrow \alpha \to \beta \in T$$

The *order* of base types $T_0$ is 1. The order of function types is the maximum of one plus the order of the argument type and the order of the result type. The order of a term is defined as the order of its type.

We are now ready to define our class of patterns, the *deterministic second-order patterns.* As we will see later, matching a pattern in this class with a closed term yields at most one match.

**Definition 1 ($\mathcal{DSP}$)** *A term $P$ is said to be a deterministic second-order pattern ($\mathcal{DSP}$), if the arguments $E_1, \ldots, E_m$ of any free variable occurring in the pattern satisfy the following conditions.*
(i) $\forall i.\, FV(E_i) \neq \{\,\}$
(ii) $\forall i, j.\, i \neq j \Rightarrow E_i \ntrianglelefteq E_j$
(iii) $\forall i.\, (v \in FV(E_i) \Rightarrow v \notin FV(P))$
(iv) *For all $i$, $E_i$ is first-order.*

$\square$

The conditions on the arguments are relaxation of Miller's idea from "distinct and bound variables" to "non-mutually embedded terms containing bound variables": (i) $E_i$ should not be a closed term. For example, the term $p\ 1$ is not a $\mathcal{DSP}$ because the argument 1 is closed. (ii) For all $i, j (i \neq j)$, $E_i$ is not a subterm of $E_j$. Therefore, $\lambda x.\ p\ x\ (x + 1)$ is not a $\mathcal{DSP}$ since the argument $x$ is a subterm of another argument $x + 1$. (iii) Free variables in $E_i$ should be bounded in the pattern

$P$. As a result, $p\ q$ is not $\mathcal{DSP}$. (iv) For example, $p\ (\lambda x.\ x)$ is not $\mathcal{DSP}$ because the argument $(\lambda x.\ x)$ is more than first-order.

The following is examples of $\mathcal{DSP}$ where $c$ and $d$ are constants.

$$\lambda x.\, p\ (c\ x)\ (d\ x)$$
$$\lambda x\, y.\, p\ x\ (c\ y)$$
$$\lambda x.\, c\ (p\ x)\ (q\ x)$$

In the rest of the paper, we use the following notational convention. Small letters $a$, $b$, $c$, $d$ represent constants, and other small letters such as $p$, $q$, $v$, $x$, $y$, $z$ represent variables. Normally, we use $p, q$ to denote the free variables and $x, y, z$ to denote bound variables. Greek identifiers $\phi$, $\psi$, $\sigma$ represent matches (substitutions), and capital letters represent terms or patterns. Lists of variables $x_1 \cdots x_l$ are represented by $\bar{x}$, and lists of terms $E_1 \cdots E_m$ by $\bar{E}$. For example, a term $\lambda x_1 \cdots x_l.\ p\ E_1 \cdots E_m$ is represented by $\lambda \bar{x}.\ p\ \bar{E}$.

## 3. Deterministic Second-order Matching

A *pattern* is a term which can contain free variables. Given a pattern $P$ and a closed term $T$ where $P$ and $T$ are $\beta\eta$-normal, a *rule* is a pair of terms written as $P \rightarrow T$.

The general *matching problem* is: given a rule $P \rightarrow T$, find all the substitutions $\phi$ such that $\phi\ P =_{\alpha\beta\eta} T$. Such a substitution $\phi$ is called a *match*, denoted by $\phi \vdash P \rightarrow T$. If there exists at most one match $\phi$, we say the matching is *deterministic.* If there exists exactly one match, we simply say that the match $\phi$ is *unique.* If the maximum order of the free variables in $P$ is at most two, we say that matching problem is *second-order.*

Second-order matching is known to be nondeterministic. Algorithms computing all the matches has been proposed in, for example, [9]. The contribution of this paper, on the other hand, is to show that second-order matching is deterministic if we restrict the patterns to $\mathcal{DSP}$.

To begin with, let us introduce the important concept of discharging subterms. Discharging $E_1, \ldots, E_m$ by $y_1, \ldots, y_m$ in $T$ means replacement of all the occurrences of $E_1, \ldots, E_m$ with fresh variables $y_1, \ldots, y_m$ respectively in $T$. One possible implementation is given in Fig. 1. Intuitively, the function

```
discharge s c  =  c
discharge s v  =  replace s v
discharge s (λx. T₁)  =
  let T' = replace s (λx. T₁)
  in if T' = (λx. T₁) then λx. (discharge s T₁)
       else T'
discharge s (T₁ T₂)  =
  let T' = replace s (T₁ T₂)
  in if T' = (T₁ T₂)
    then ((discharge s T₁) (discharge s T₂))
    else T'

replace [] T  =  T
replace ((y, E) : s) T  =
  if E = T then y else replace s T
```

Fig. 1. Discharging Algorithm

$$\textit{discharge}\ [(y_1, E_1), \ldots, (y_m, E_m)]\ T$$

replaces all the occurrences of $E_1, \ldots, E_m$ with fresh variables $y_1, \ldots, y_m$ respectively in $T$. That is:

$$B = \textit{discharge}\ [(y_1, E_1), \ldots, (y_m, E_m)]\ T$$
$$\Rightarrow (\lambda \bar{y}.\, B)\ \bar{E} =_{\alpha\beta\eta} T\ \wedge\ \forall i.\, E_i \ntrianglelefteq B.$$

**Lemma 2** *If $P = \lambda \bar{x}.\ p\ \bar{E}$ is a $\mathcal{DSP}$ where $p$ is a free variable, then there is at most a single match $\phi$ such that $\phi \vdash P \rightarrow T$.*

**PROOF.** There is no match if $T$ is not transformed into $\lambda \bar{x}.\ T'$ by $\alpha\eta$-conversion. The match of a rule $p\ \bar{E} \rightarrow T'$ should be in the form $\{p \mapsto \lambda \bar{y}.\, B\}$. Since free variables in each $E_i$ are bounded in $P$ by definition 1.(iii), by definition of match the equation $(\lambda \bar{y}.\, B)\ \bar{E} =_{\alpha\beta\eta} T'$ should be satisfied. Therefore, a term $B$ is a result of replacing

$\bar{E}$ with $\bar{y}$ in $T'$. By definition 1.(i), subterms $E_i$ $(1 \leq i \leq m)$ contain free variables and if we leave any occurrences of $E_i$ in $B$, then $\lambda\bar{y}.\,B$ will contain free variables. This results in generating illegal substitution containing free variables. Instead, a term $B$ should be obtained by full discharging; replacing all the occurrences of $\bar{E}$ with $\bar{y}$ in $T'$, i.e, $(\lambda\bar{y}.\,B)\ \bar{E} =_{\alpha\beta\eta} T' \ \wedge\ \forall i.\, E_i \not\trianglelefteq B$. If some free variables still occur in $B$ after the discharging, this results in illegal substitution. Otherwise, since one argument is not a subterm of another argument by definition 1.(ii), the order of replacing does not affect the result of the match. Thus, the match is obtained deterministically. $\square$

Note that as in the proof, for discharging the arguments of free variables in a $\mathcal{DSP}$, we can use any discharging function satisfying the condition $(\lambda\bar{y}.\,B)\ \bar{E} =_{\alpha\beta\eta} T' \ \wedge\ \forall i.\, E_i \not\trianglelefteq B$. In the following, we use the function *discharge* for discharging the arguments from a term. We now give our main theorem below.

**Theorem 3** *If $P$ is a $\mathcal{DSP}$, there is at most a single match $\phi$ such that $\phi \vdash P \to T$.*

**PROOF.** We use mathematical induction on the structure of the pattern.

Case ($P = \lambda\bar{x}.\ c\ \bar{E}$). There is no match if the corresponding term cannot be transformed into $\lambda\bar{x}.\ c\ \bar{F}$ by $\alpha\eta$-conversion where the lengths of $\bar{E}$ and $\bar{F}$ are equal. Otherwise, the matching can be decomposed into $m$ matchings $\phi_i \vdash \lambda\bar{x}.\ E_i \to \lambda\bar{x}.\ F_i$ for $i = 1 \ldots m$. By the induction hypothesis, each match $\phi_i \vdash \lambda\bar{x}.\ E_i \to \lambda\bar{x}.\ F_i$ is unique or there is no match in which case $\phi_i = \mathit{fail}$. Therefore $\phi' \vdash P \to T$ is the unique match or there is no match if $\phi'$ is *fail* where $\phi' = \phi_1 \circ \cdots \circ \phi_m$.

Case ($P = \lambda\bar{x}.\ v\ \bar{E} \wedge v \notin FV(P)$). Similar to the first case.

Case ($P = \lambda\bar{x}.\ v\ \bar{E} \wedge v \in FV(P)$). By Lemma 2, the match generated by the pattern is unique or there is no match. $\square$

For example, consider $P = \lambda x.\ p\ (c\ x)\ (d\ x)$ and the term $T = \lambda x.\ a\ (c\,x)\ (b\ (d\ x))$ where $a$, $b$, $c$ and $d$ are constants , $p$ and $x$ are variables, and $p$ occurs free in $P$. To match $P$ against $T$, we replace $c\,x$ and $d\,x$ with fresh variables $y_1$ and $y_2$ in $T$ resulting in the unique match $\{p \mapsto \lambda y_1\, y_2.\ a\ y_1\ (b\ y_2)\}$.

## 4. An Efficient Deterministic Second-order Matching Algorithm

Given a rule $P \to T$ where $P$ is a $\mathcal{DSP}$, the algorithm $\mathcal{M}[\![P \to T]\!]$, defined in Fig. 2 computes its unique match if it exists. Otherwise it returns the special match fail. For example, $\mathcal{M}[\![c \to \lambda x.\, d]\!]$ returns *fail*. In Fig. 2, the first case acts as $\eta$-expansion, so, $\mathcal{M}[\![\lambda x.\, p\ (c\ x) \to c]\!]$ returns $\mathcal{M}[\![\lambda x.\, p\ (c\ x) \to \lambda x.\, c\ x]\!]$. The second and the third cases correspond to the cases in our proof of Theorem 3. If the heads of the pattern and the term are equal and the lengths of their arguments are the same, the rule is decomposed into smaller ones. The fourth case which calls the function *discharge* for exhaustive discharging corresponds to Lemma 2. $\mathcal{M}[\![\lambda a\, r.\, a \otimes \mathit{sum}\ r \to \lambda a\, r.\, a * a + \mathit{sum}\ r]\!]$ is an example of the fourth case and computes the following match.

$$\{\otimes \mapsto \lambda x\, y.\, \mathit{discharge} \quad [(y_1, a), (y_2, \mathit{sum}\ r)]\ (a * a + \mathit{sum}\ r)\}$$

Formally, we can prove the soundness of the algorithm $\mathcal{M}$, i.e., $\mathcal{M}$ returns the unique match if there exists one.

**Theorem 4 (Soundness)** *If $P$ is a $\mathcal{DSP}$, then*

$$\phi \vdash P \to T \Leftrightarrow \phi = \mathcal{M}[\![P \to T]\!] \wedge \phi \neq \mathit{fail}$$

**PROOF.** We prove it by induction on the structure of the pattern. The proof is straightforward except for the case where the rule is in the form $\lambda\bar{x}.\, p\, E_1\ \cdots\ E_m \to \lambda\bar{x}.\, T_1$. We only show this case.

($\Leftarrow$) Let

$$B = \mathit{discharge}\ [(y_1, E_1), \ldots, (y_m, E_m)]\ T_1$$

By the property of *discharge*, $(\lambda\bar{y}.\ B)\ \bar{E} = T_1$ holds. Therefore, the following matching property holds.

$$\{p \mapsto \lambda\bar{y}.\ B\} \vdash \lambda\bar{x}.\ p\ \bar{E} \to \lambda\bar{x}.\ T_1$$

$$
\begin{aligned}
&\mathcal{M}[\![\lambda x_1 \cdots x_l.\ P_1 \to \lambda x_1 \cdots x_o.\ T_1]\!] \ = \\
&\quad \mathcal{M}[\![\lambda x_1 \cdots x_l.\ P_1 \to \lambda x_1 \cdots x_l.\ T_1\ x_{o+1} \cdots x_l]\!] \\
&\qquad \textbf{if } P_1 \text{ and } T_1 \text{ are not } \lambda\text{-abstraction} \\
&\mathcal{M}[\![\lambda \bar{x}.\ c\ E_1\ \cdots\ E_m \to \lambda \bar{x}.\ d\ T_1\ \cdots\ T_m]\!] \ = \\
&\quad \mathcal{M}[\![\lambda \bar{x}.\ E_1 \to \lambda \bar{x}.\ T_1]\!] \circ \cdots \circ \mathcal{M}[\![\lambda \bar{x}.\ E_m \to \lambda \bar{x}.\ T_m]\!] \\
&\qquad \textbf{if } c = d \\
&\mathcal{M}[\![\lambda \bar{x}.\ x_i\ E_1\ \cdots\ E_m \to \lambda \bar{x}.\ x_j\ T_1\ \cdots\ T_m]\!] \ = \\
&\quad \mathcal{M}[\![\lambda \bar{x}.\ E_1 \to \lambda \bar{x}.\ T_1]\!] \circ \cdots \circ \mathcal{M}[\![\lambda \bar{x}.\ E_m \to \lambda \bar{x}.\ T_m]\!] \\
&\qquad \textbf{if } i = j \\
&\mathcal{M}[\![\lambda \bar{x}.\ p\ E_1\ \cdots\ E_m \to \lambda \bar{x}.\ T_1]\!] \ = \\
&\quad \{p \mapsto \lambda y_1 \cdots y_m.\, B\} \\
&\qquad \textbf{if } \lambda y_1 \cdots y_m.\, B \text{ is closed} \\
&\qquad \textbf{where} \\
&\qquad y_1, \ldots, y_m \text{ are fresh variables} \\
&\qquad B = \mathit{discharge}\ [(y_1, E_1), \ldots, (y_m, E_m)]\ T_1 \\
&\mathcal{M}[\![\_]\!] \ = \ \mathit{fail}
\end{aligned}
$$

Fig. 2. The Matching Algorithm

($\Rightarrow$) By theorem 3, there is at most a single match $\phi$ such that

$$\phi \vdash \lambda \bar{x}.\ p\ E_1\ \cdots\ E_m \to \lambda \bar{x}.\ T_1$$

The form of the match should be $\phi = \{p \mapsto \lambda y_1 \cdots y_m.\, B\}$ where

$$\{y_1 \mapsto E_1, \ldots, y_m \mapsto E_m\}\ B =_{\alpha\beta\eta} T_1$$

A term $B$ should be made by replacing some $E_i$ with $y_i$ from $T_1$. By the definition 1.(i), $E_i$ contains free variables. Thus if $B$ contains $E_i$, then $\phi$ is illegal match. Therefore a term $B$ should be made by replacing all the occurence $E_i$ with $y_i$ from $T_1$. This operation matches $B = \mathit{discharge}\ [(y_1, E_1), \ldots, (y_m, E_m)]\ T_1$. $\square$

The complexity of our matching algorithm is summarized in the following theorem. Let $\mathit{size}(t)$ be a function computing a size of the term $t$.

$$
\begin{aligned}
\mathit{size}\ c &= 1 \\
\mathit{size}\ v &= 1 \\
\mathit{size}\ (t_1\ t_2) &= 1 + \mathit{size}\ t_1 + \mathit{size}\ t_2 \\
\mathit{size}\ (\lambda x.\, t) &= 1 + \mathit{size}\ t
\end{aligned}
$$

**Theorem 5 (Efficiency)** *Let $P$ be a $\mathcal{DSP}$, $n$ be the size of the term $T$, and $m$ be the size of the pattern $P$. The time complexity of $\mathcal{M}[\![P \to T]\!]$ is* $\mathrm{O}(m^2 n)$.

**PROOF.** Except for the second last case of the definition of the matching algorithm $\mathcal{M}$ in Fig. 2, the time complexity of $\mathcal{M}$ is straightforwardly linear in the size of the pattern. For the second last case, the function *discharge* traverses the term, calling the function *replace* that checks for each argument $E_i$. Since equality check in *replace* needs $\mathrm{O}(m)$, *replace* costs $\mathrm{O}(m^2)$. Therefore *discharge* costs $\mathrm{O}(m^2 n)$. $\square$

Since $m$ is often small and bounded, and $m$ is much smaller than $n$ in practice, the algorithm is almost $\mathrm{O}(n)$. For a fixed pattern, the algorithm is $\mathrm{O}(n)$.

## 5. Conclusion

In this paper, we proposed a class of patterns that have the unique second-order match. We believe that the advantage of determinism is helpful for user to express his intension to the compiler of program transformation system in a more precise and predictable way. And it makes possible for the second-order matching to be used in functional languages efficiently [7].

Our pattern is a simple and natural extension of Miller's pattern [11] which has a single most general unifier, and is a sort of a restriction of the two-step valid pattern of Sittampalam and de Moor's [14,15]. But it is not linear time. They also developed efficient higher-order matching algorithm, one-step matching algorithm which covers at least complete second-order matches [5,14].

While the second-order matching algorithm is NP-complete [1] and the implementations are expensive [3,9], the restriction on patterns sometimes leads fast matching algorithms. Second order pure matching (even unification) with bounded number of variables is PTIME [16]. Hirata, Yamada and Harao [8] have studied the complexity of various second-order matching. According to their classification, $\mathcal{DSP}$ is *predicate*, namely any arguments of free variables includes no function variables. The matching problem of a predicate is polynomial if it is *binary function-free*, namely, any function variables are at most 2-ary and it includes no function constants. *Linear context matching*, a restricted form of linear higher-order matching, is $O(n^3)$ [13]. They solve the problem by dynamic programming with the table of the size $O(n^2)$ building from the bottom up. Our restriction makes our matching algorithm fast; given a fixed pattern, the time complexity of our deterministic matching algorithm is linear in the size of a term being matched.

## References

[1] L. Baxter, The complexity of unification, Ph.D. thesis, Department of Computer Science, University of Waterloo (1977).

[2] R. Bird, Introduction to Functional Programming using Haskell (second edition), Prentice Hall, 1998.

[3] R. Curien, Z. Qian, H. Shi, Efficient second-order matching, in: H. Ganzinger (Ed.), Rewriting Techniques and Applications, Vol. 1103 of LNCS, Springer, New Brunswick, New Jersey, USA, 1996, pp. 317–331.

[4] O. de Moor, G. Sittampalam, Generic program transformation, in: Third International Summer School on Advanced Functional Programming, Vol. 1608 of LNCS, Springer-Verlag, Braga, Portugal, 1998, pp. 116–149.

[5] O. de Moor, G. Sittampalam, Higher order matching for program transformation, in: A. Middeldorp, T. Sato (Eds.), Functional and Logic Programming, 4th Fuji International Symposium, Vol. 1722, Springer, Tsukuba, Japan, 1999, pp. 209–224.

[6] O. de Moor, G. Sittampalam, Higher-order matching for program transformation, Theoretical Computer Science 269 (1–2) (2001) 135–162.

[7] R. Heckmann, A functional language for the specification of complex tree transformation, in: Proc. ESOP, Vol. 300 of LNCS, 1988, pp. 175–190.

[8] K. Hirata, K. Yamada, M. Harao, Tractable and intractable second-order matching problems, in: T. Asano, H. Imai, D. T. Lee, S. Nakano, T. Tokuyama (Eds.), Computing and Combinatorics, 5th Annual International Conference, Vol. 1627 of LNCS, Springer, Tokyo, Japan, 1999, pp. 432–441.

[9] G. P. Huet, B. Lang, Proving and applying program transformations expressed with second-order patterns, Acta Informatica 11 (1978) 31–55.

[10] B. Krieg-Brückner, J. Liu, H. Shi, B. Wolff, Towards correct, efficient and reusable transformational developments, in: M. Broy, S. Jähnichen (Eds.), KORSO — Methods, Languages, and Tools for the Construction of Correct Software, Vol. 1009 of LNCS, Springer-Verlag, 1995, pp. 270–284.

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

[12] A. Schubert, Linear interpolation for the higher-order matching problem, in: M. Bidoit, M. Dauchet (Eds.), Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Vol. 1214, Springer, Lille, France, 1997, pp. 441–452.

[13] M. Schmidt-Schauß, J. Stuber, On the complexity of linear and stratified context matching problems, Rapport de recherche RR-4923 (Jul. 2003).

[14] G. Sittampalam, Higher-order matching for program transformation, Ph.D. thesis, University of Oxford (2001).

[15] G. Sittampalam, O. de Moor, Higher-order pattern matching for automatically applying fusion transformations, in: O. Danvy, A. Filinski (Eds.), 2nd Symposium on Programs as Data Objects, Vol. 2053 of LNCS, Springer-Verlag, Aarhus, Denmark, 2001, pp. 218–237.

[16] T. Wierzbicki, Complexity of the higher order matching, in: H. Ganzinger (Ed.), Automated Deduction, Vol. 1632 of LNCS, Springer, Trento, Italy, 1999, pp. 82–96.