# Deterministic Second-order Patterns in Program Transformation

Tetsuo YOKOYAMA[1], Zhenjiang HU[1,2], and Masato TAKEICHI[1]

[1] Department of Mathematical Informatics, Graduate School of Information Science and Technology, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, JAPAN
`yokoyama@ipl.t.u-tokyo.ac.jp, {hu,takeichi}@mist.i.u-tokyo.ac.jp`
[2] PRESTO21, Japan Science and Technology Corporation

**Abstract.** Higher-order patterns, together with higher-order matching, enable concise specification of program transformation, and have been implemented in several program transformation systems. However, higher-order matching generally generates nondeterministic matches, and the matching algorithm is so expensive that even second-order matching is NP-complete. It is orthodox to impose constraint on the form of 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, while unification of general patterns is nondeterministic (and even undecidable). 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 pattern* covers a wide class of useful patterns for program transformation. Our deterministic matching algorithm is as fast as the first-order matching algorithm, almost in proportion to the size of the term.

**Keywords**: Higher-order pattern matching, Functional Programming, Program Transformation, Fusion Transformation.

## 1 Introduction

Patterns, together with pattern matching algorithms, play an important role in specification of transformation rules as well as implementation of transformation systems. Usually, the more flexible the patterns are, the more difficult it is to design efficient matching algorithms. The first-order patterns are simple and the first-order matching algorithms are cheap and deterministic, but the first-order patterns lack descriptive power. In contrast, the second- (or higher-) order patterns [1–3] are flexible enabling concise specification of powerful transformation, but the second-order matching algorithms are expensive and nondeterministic.

Consider, as an example, the (cheap) fusion transformation [4, 5], which is to optimise programs by eliminating unnecessary intermediate data structures passed between functions. The basic fusion transformation rule (in Haskell like notation [6]) is:

$$\frac{a \otimes f\ r = f(a \oplus r)}{f\ .\ \mathit{foldr}\ (\oplus)\ e = \mathit{foldr}\ (\otimes)\ (f\ e)}$$

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 fusible condition, namely $a \otimes f\ r = f\ (a \oplus r)$ holds. However, specification of this simple transformation rule with the first-order pattern has proved to be a challenge [7].

Notice that the key step in fusion transformation is to find a function $\otimes$ meeting the fusible condition. This is actually a higher-order matching problem: matching the higher order pattern $\lambda a\, r.\, a \otimes f\ r$ with the term[3] $\lambda a\, r.\, f\ (a \otimes r)$ to obtain a substitution (definition) for the pattern variable $\otimes$. To be more concrete, let us see the fusion of the following program for computing the sum of squares of each element of a list[4].

$$\begin{array}{lcl} \mathit{sumsq} & = & \mathit{sum}\ .\mathit{foldr}\ (\lambda a\, r.\, a * a : r)\ [\,] \\ \quad \textbf{where}\ \mathit{sum}\ [\,] & = & 0 \\ \qquad\quad \mathit{sum}\ (a : x) & = & a + \mathit{sum}\ x \end{array}$$

With the fusion rule, we can obtain $\mathit{sumsq} = \mathit{foldr}\ (\otimes)\ 0$, if we can find a function $\otimes$ such that the pattern $\lambda a\, r.\, a \otimes \mathit{sum}\ r$ can match with the term $\lambda a\, r.\, a{*}a{+}\mathit{sum}\ r$, the normalised form of $\lambda a\, r.\, \mathit{sum}\ (a \oplus r)$ where $\oplus = \lambda a\, r.\, a * a : r$. Applying the second-order matching algorithm immediate yields the match:

$$\{\otimes \mapsto \lambda x\, y.\, x * x + y\}.$$

It is worth noting that while the second-order matching is powerful enough to find the solution, the first-order matching will give no solution.

Despite the attractive power of higher-order patterns and higher-order matching, there are several significant objections to the use of higher-order matching for implementing program transformation, particularly in functional languages.

- First, higher-order matching is known to be so expensive that even second-order is NP-complete [8]. Therefore, a real efficient implementation is out of the question.
- Second, higher-order matching algorithms are generally nondeterministic, resulting in more than one solution. Unlike logic languages such as Prolog which deal with this sort of nondeterminism by means of backtracking, it cannot be directly handled by functional languages.
- Lastly, although there is a clear specification of the solutions to the matching problem, nondeterminism makes semantics complex and it is often difficult to explain why a particular match was not produced.

[3] Strictly speaking, the term should be normalised before being matched with a pattern.

[4] Here binary operators (.), $(*)$, (:), $(+)$ are constants.

Fortunately, the experience of implementing program transformation systems tells that it is not really necessary to have full flexibility of higher-order patterns and higher order matching in practice. We may, therefore, think of imposing reasonable restriction on the form of patterns to generate desirable higher-order matching that is both *deterministic* and *efficient*. It is known that by restricting the form of patterns, one can make possibly undecidable fifth-order matching be decidable [9], and infinite third-order matching be finite [3].

We shall focus on the second-order matching [1], a useful matching particularly useful for program transformation, and we aim to provide a reasonable restriction on the second-order patterns so as to make the second order matching be not only deterministic and efficient, but also powerful enough to specify transformation rules. As a matter of fact, in the context of unification, Miller has defined a class of *deterministic higher-order pattern* [10]. In his higher-order patterns, each occurrence of free variable should be applied to a sequence of distinct bound variables. For example, the pattern

$$\lambda x\, y.\, p\ y\ x$$

is valid, since the free variable $p$ appears at the head of the application $p\ y\ x$ and the arguments of $p$, namely $x$ and $y$, are distinct bound variables. However, the patterns of $\lambda x\, y.\, p\ x\ x$ where $p$ has the same bound variable $x$, $\lambda x\, y.\, p\ (x + x)\ y$ where the argument $(x + x)$ of $p$ is not a variable, and $\lambda x\, y.\, p\ q\ x$ where the argument $q$ of $p$ is not a bound variable, are all invalid. It has been proved that the general higher-order unification (matching) with respect to these patterns is deterministic, and an implementation has been given [11]. Miller's patterns are, however, too restrictive to describe program transformation rules. For instance, the arguments of free variables in a pattern may be complicated terms instead of variables, as seen in the fusion law where the arguments of the free variable $\otimes$ in the pattern are $a$ and $f\ r$.

In this paper, we relax the restriction of Miller's higher-order patterns by allowing the arguments to be terms, and propose a class of deterministic second-order patterns, called $\mathcal{DSP}$, with the following features.

- $\mathcal{DSP}$ covers a wider class of patterns that are often used in specification of program transformation and program calculation. It enables concise description of the fusion law. This is very important, the fusion law plays an essential role in program transformation and calculation [12, 13] and many other transformations can be formalised using the fusion law.
- $\mathcal{DSP}$ leads to a deterministic and efficient algorithm for second-order matching. Our main idea of $\mathcal{DSP}$ is the deterministic choice of *discharging*, with which the second-order matching problem is boiled down to the first-order one. It is our hope that our approach would provide a new and effective way to incorporate higher-order patterns into functional languages [14].

The organisation of the paper is as follows. In Sect. 2, We give a formal definition of our deterministic second-order patterns. In Sect. 3, we prove the determinism of our matching algorithm with respect to deterministic second-order patterns. In Sect. 4, we present an efficient algorithm, and prove soundness and efficiency of the algorithm. In Sect. 6, we briefly explain related works and conclude the paper.

## 2 Deterministic Second-order Patterns

We consider simply-typed lambda *terms* defined as follows.

$$\begin{array}{lll} T = & c & \{\text{ constant }\} \\ \mid & v & \{\text{ variable }\} \\ \mid & T\ T & \{\text{ application }\} \\ \mid & \lambda x\,.\,T & \{\text{ lambda abstraction }\} \end{array}$$

Let $FV$ be a function mapping from a term to a set of free variables in the term. For example, $FV(\lambda x.\, p\ x)$ returns $p$. We call a term $E$ *closed* if $FV(E) = \{\,\}$. For readability we sometimes use the infix notation, so $x + y$ denotes the term $(+)\ x\ y$. We call a term *β-normal* if the term does not contain any $\beta$-redex, and we call term *η-normal* if the term does not contain any $\eta$-redex. We sometimes write $\lambda\bar{x}.\, p\ \bar{E}$ for $\lambda x_1 \cdots \lambda x_l.\, p\ E_1 \cdots E_m$.

We say that a term $E_1$ is a *subterm* of $E_2$, denoted by $E_1 \trianglelefteq E_2$, if $E_1 \in \mathit{subTerm}(E_2)$ (here $\alpha$-renaming is implicitly assumed), where *subTerm* is defined below.

$$\begin{array}{ll} \mathit{subTerm}(c) & = \{c\} \\ \mathit{subTerm}(v) & = \{v\} \\ \mathit{subTerm}(E_1\ E_2) & = \{E_1\ E_2\} \cup \mathit{subTerm}(E_1) \cup \mathit{subTerm}(E_2) \\ \mathit{subTerm}(\lambda x.\, E) & = \{\lambda x.\, E\} \cup \mathit{subTerm}(E) \end{array}$$

Let $v\ t_1\ \cdots\ t_n$ be a subterm and $v$ be a variable. We call $t_1, \ldots, t_n$ be *arguments* of $v$, and call $v$ *head* of the subterm.

We write *substitutions* (or called *matches*) as a mapping from variables to closed terms like

$$\phi = \{p \mapsto \lambda x.\, x\ b\}.$$

We denote domain of substitution $\phi$ as $\mathit{dom}(\phi)$ and range of the substitution as $\mathit{range}(\phi)$. The composition of substitutions $\phi$ and $\psi$ is defined if the substitutions are *compatible*, i.e., the same variables in domains do not have different 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 $\alpha\beta\eta$-conversion. Otherwise, it will return the special match $fail$. Note that $fail$ is the zero unit of match composition, that is, $fail\ .\ m = m\ .\ fail = fail$. For example, the composition of substitutions $\{p \mapsto c\}\ .\ \{p \mapsto \lambda x.\, x\}$ is $fail$.

*Types* are constructed in the usual way in simply typed lambda calculus. Let $T_0$ be a set of base type. A type set $T$ are defined as follows.

$$\begin{array}{l} \alpha \in T_0 \quad \Rightarrow \alpha \in T \\ \alpha, \beta \in T \Rightarrow \alpha \rightarrow \beta \in T \end{array}$$

The *order* of a type $\tau$ $ord(\tau)$ is defined as follows.

$$\begin{array}{lll} ord(\alpha) & = 1, & \textbf{if } \alpha \in T_0 \\ ord(\alpha \rightarrow \beta) & = max\{ord(\alpha)+1, ord(\beta)\} & \end{array}$$

The order of any base types is 1. The order of a 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.

Terms with free variables are called *patterns*. 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 $P \rightarrow T$.

We are now ready to define our class of patterns. We call them *deterministic second-order patterns*, because as we will see later matching a pattern belonging to this class with a closed term will give at most one match. The class of patterns is a simple extension of Miller's higher-order patterns which has at most single unification; the arguments of every free variables in the pattern must be distinct and bound variables.

**Definition 1 (Deterministic Second-order Patterns).** A term $P$ is said to be a deterministic second-order pattern $\mathcal{DSP}$, if (1) *maximum* $\{ord(v) \mid v \in FV(P)\} \leq 2$, and (2) 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 $\forall i.\, v \in FV(E_i) \Rightarrow v \notin FV(P)$ □

The condition (1) simply requires that free variables in the pattern should have the order of at most 2, due to our focus on the second-order case. It follows that each $E_i$ should be a first-order term; $p\ (\lambda x.\, x)$ is not a valid pattern because the argument is a function which is not a first-order term. The condition (2) on the arguments is a 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, so the term $p\ 1$ is not a valid pattern because the argument 1 of the free variable $p$ is a closed term, which does not contain any free variable; (ii) For all $i, j (i \neq j)$, $E_i$ is not a subterm of $E_j$, so $\lambda x.\, p\ x\ (x+1)$ is not a valid pattern since the argument $x$ is a subterm of another argument $x+1$; (iii) $E_i$ should not contain any pattern (free) variable, i.e., free variables in $E_i$ should not be free in the pattern $P$, so $p\ q$ is not a valid pattern.

$\mathcal{DSP}$ covers a wide class of useful patterns for describing transformation rules. The pattern $\lambda a\, r.\, a \otimes f\ r$ in the fusion law discussed in the introduction is a $\mathcal{DSP}$. Another example is, the following $\mathcal{DSP}$ pattern

$$\lambda w\, x.\, \textbf{if}\ p\ x\ \textbf{then}\ q\ x\ \textbf{else}\ r\ (Car\ x)\ (w\ (Cdr\ x))$$

which is used to extract a specific program structure. This pattern is beyond Miller's higher-order patterns since the arguments of free variable $r$ are not variables. Matching this pattern with the program

$$\begin{array}{l}\lambda \mathit{reverse}\, x.\, \textbf{if}\ \mathit{Null}\ x\ \textbf{then}\ x\ \textbf{else}\\ \qquad \mathit{Append}\ (\mathit{reverse}\ (\mathit{Cdr}\ x))\ (\mathit{Cons}\ (\mathit{Car}\ x)\ \mathit{Nil})\end{array}$$

gives the unique match

$$\{p \mapsto \mathit{Null}, q \mapsto \lambda x.\, x, r \mapsto \lambda z\, w.\, \mathit{Append}\ w\ (\mathit{Cons}\ z\ \mathit{Nil})\},$$

and matching the pattern with the program

$$\begin{array}{l}\lambda \mathit{mapsquare}\, x.\, \textbf{if}\ \mathit{Null}\ x\ \textbf{then}\ x\ \textbf{else}\\ \qquad \mathit{Cons}\ (\mathit{square}\ (\mathit{Car}\ x))\ (\mathit{mapsquare}\ (\mathit{Cdr}\ x))\end{array}$$

gives another but unique match

$$\{p \mapsto \mathit{Null}, q \mapsto \lambda x.\, x, r \mapsto \lambda z\, w.\, \mathit{Cons}\ (\mathit{square}\ z)\ w\}.$$

We will discuss the matching algorithm later.

In the rest of the paper, we will use the following notational convention. We will use $a$, $b$, $c$, $d$ and ones starting from capital letters to represent constants, use other small letters such as $p$, $q$, $v$, $x$, $y$, $z$ represent variables. To distinguish free variables and bound variables in a pattern, we will use $p, q$ to denote the free variables and $x, y, z$ to denote bound variables. We will use the Greek identifiers $\phi$, $\psi$, $\sigma$ to represent matches (environments), and the capital letters to represent terms, or patterns.

## 3 Deterministic Second-order Matching

So far we have given the definition of our patterns. We call it deterministic second-order pattern, because matching a term with this pattern will guarantee to produce a unique match if there exists. We prove this in this section, and show how to do this matching efficiently in Sect. 4.

The general *matching problem* is defined as follows. Given a rule $P \to T$, find all the substitutions $\phi$ such that $\phi\ P =_{\alpha\beta\eta} T$. We call this substitution *match*, and write

$$\phi \vdash P \to T$$

to indicate that $\phi$ is a match of the rule $P \to T$. If a matching produces at most one $\phi$ such that $\phi \vdash P \to T$, we say it is *deterministic*, or simply say that $\phi \vdash P \to T$ is deterministic. If the free variables in $P$ have the order of more than one, we say that the matching problem is of *higher-order*. Generally, a higher-order matching returns infinite solutions and may be undecidable and incomplete. It is known that the second-order matching is nondeterministic, although there exists a complete matching algorithm [1] computing all the matches.

```
$discharge\ s\ c\ =\ c$
$discharge\ s\ v\ =\ replace\ s\ v$
$discharge\ s\ (\lambda x.T_1)\ =$
  $\mathbf{let}\ T' = replace\ s\ (\lambda x.T_1)$
  $\mathbf{in\ if}\ T' = (\lambda x.T_1)\ \mathbf{then}\ \lambda x.(replace\ s\ T_1)\ \mathbf{else}\ T'$
$discharge\ s\ (T_1\ T_2)\ =$
  $\mathbf{let}\ T' = replace\ s\ (T_1\ T_2)$
  $\mathbf{in\ if}\ T' = (T_1\ T_2)\ \mathbf{then}\ ((discharge\ s\ T_1)\ (discharge\ s\ T_2))\ \mathbf{else}\ T'$

$replace\ []\ T_1\ =\ T_1$
$replace\ ((y, E) : s)\ T_1\ =\ \mathbf{if}\ E = T_1\ \mathbf{then}\ y\ \mathbf{else}\ replace\ s\ T_1$
```

**Fig. 1.** Discharging Algorithm

We call a transformation ($discharge\ [(y_1, E_1), \ldots, (y_m, E_m)]\ T$) *discharging*, or *discharging* $E_1, \ldots, E_m$ from $T$ by $y_1, \ldots, y_m$ where the function *discharge* is defined as Fig. 1. Note that ($discharge\ [(y_1, E_1), \ldots, (y_m, E_m)]\ T$) does not contain $E_1, \ldots, E_m$ as subterms. For example, discharging $1 : []$ from $1 : 1 : []$ by $y$ results in $1 : y$, and discharging $x : []$ from $1 : x : []$ by $y$ results in $1 : y$. Note that the above definition of discharging is only syntactical replacement.

If we restrict patterns to be $\mathcal{DSP}$, the second-order matching becomes deterministic, returning at most one match for any rule. We start with a simple case.

**Lemma 2.** Let $P_1 = \lambda x_1 \cdots x_l.p\ E_1\ \cdots\ E_m$ be a $\mathcal{DSP}$ where $p \in FV(P_1)$, and $T_1$ be an arbitrary closed term. Then, $\phi \vdash P_1 \rightarrow T_1$ is deterministic.

*Proof.* There is no match if $T_1$ is not in the form of $\lambda x_1 \cdots x_l.T_2$ (after $\alpha\beta\eta$-conversion). To obtain a target of a mapping from $p$ of a match of a rule $\lambda x_1 \cdots x_l.p\ E_1\ \cdots\ E_m \rightarrow \lambda x_1 \cdots x_l.T_2$, we have to discharge all the occurrences of $E_1, \ldots, E_m$ in $T_2$ by $y_1, \ldots, y_m$ exhaustively and enclosing it by $y_1, \ldots, y_m$, since $E_i(1 \leq i \leq m)$ contains free variables (Def.1.(2)i) and if we leave some occurrences in $T_2$ then we will not be able to find a map from $p$ to a *closed* term. For a $\mathcal{DSP}$, syntactical replacement is sufficient to realize discharging since the order of $E_i$ is 1(Def.1.(1)).

Moreover, since $\forall i, j.\, i \neq j \Rightarrow E_i \ntrianglelefteq E_j$(Def.1.(2)ii), the order of discharging $E_i$ does not affect the result of the match. Because of this unique way for exhaustively discharging, $\phi \vdash P_1 \rightarrow T_1$ is deterministic. □

We now give our main theorem that matching for the rule of the pattern of $\mathcal{DSP}$ and an arbitrary closed term is deterministic.

**Theorem 3 (Deterministic Second-order Matching).** If $P$ is a $\mathcal{DSP}$, $\phi \vdash P \rightarrow T$ is deterministic.

*Proof.* We prove it by induction on the structure of pattern.

Case ($P = \lambda\bar{x}.\, c\, E_1 \,\cdots\, E_m$). There is no match if the corresponding term is not transformed into $T_1 = \lambda\bar{x}.\, c\, F_1 \,\cdots\, F_m$ by $\alpha\beta\eta$-conversion. And the matching decomposed into $m$ matchings $\phi_i \vdash \lambda\bar{x}.\, E_i \rightarrow \lambda\bar{x}.\, F_i$ for $i = 1 \ldots m$. By induction hypothesis, each matching $\phi_i \vdash \lambda\bar{x}.\, E_i \rightarrow \lambda\bar{x}.\, F_i$ is deterministic, so $\phi' \vdash P \rightarrow T$ is deterministic and $\phi' = \phi_1 \circ \cdots \circ \phi_m$.

Case ($P_1 = \lambda\bar{x}.\, x_i\; E_1 \,\cdots\, E_m$). The case is similar to the first case. There is no match if the corresponding term is not transformed into $T_1 = \lambda\bar{x}.\, x_i\; F_1 \,\cdots\, F_m$ by $\alpha\beta\eta$-conversion. Then, $\phi' \vdash P_1 \rightarrow T_1$ is deterministic, since by induction hypothesis for all $i (1 \leq i \leq m)$, $\phi_i \vdash \lambda\bar{x}.\, E_i \rightarrow \lambda\bar{x}.\, F_i$ is deterministic.

Case ($P_1 = \lambda\bar{x}.\, p\; \bar{E} \wedge p \in FV(P)$). By Lemma 2, the match generated by the pattern is deterministic. □

## 4 An Efficient Deterministic Second-order Matching Algorithm

Second-order matching is generally NP-complete [8], but our restriction on patterns enables us to obtain a very efficient polynomial algorithm.

Given a rule $P \rightarrow T$ where $P$ is a $\mathcal{DSP}$, the match of a rule $P \rightarrow T$ is computed by $\mathcal{M}(P \rightarrow T)$. That is $\mathcal{M}(P \rightarrow T) \vdash P \rightarrow T$. The efficient matching algorithm $\mathcal{M}$ is defined in Fig. 2. $\mathcal{M}$ returns the unique match if it exists, otherwise returns the special match $fail$. For example, $\mathcal{M}(sum \rightarrow \lambda x.\, 0)$ returns $fail$. In Fig.2, the first case acts as $\eta$-expansion, so, $\mathcal{M}(\lambda x.\, p\; (sum\; x) \rightarrow sum)$ returns $\mathcal{M}(\lambda x.\, p\; (sum\; x) \rightarrow \lambda x.\, sum\; x)$. 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. If $m = 0$, then it returns the identity substitution, i.e., $\{\,\}$. The fourth case corresponds to Lemma 2 which calls the function *discharge* for exhaustive discharging. $\mathcal{M}(\lambda a\, r.\, a \otimes sum\; r \rightarrow \lambda a\, r.\, a * a + sum\; r)$ is the example of the fourth case and computes

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

which is

$$\{(\otimes) \mapsto \lambda x\, y.\, x * x + y)\}$$

The function *discharge* traverses the term $x * x + sum\; y$ and replaces $x$ and $sum\; y$ with $y_1$ and $y_2$ respectively. As seen in Introduction (although bound variables are renamed), the returned deterministic match is

$$\{p \rightarrow \lambda y_1\, y_2.\, y_1 * y_1 + y_2\}.$$

Formally, we can prove the soundness of the algorithm $\mathcal{M}$, i.e., if there is a match (at most one) for a rule, $\mathcal{M}$ will return the match as the result.

**Theorem 4 (Soundness).** Given a rule $P \rightarrow T$ where $P$ is $\mathcal{DSP}$, the following holds.

$$\phi \vdash P \rightarrow T \Leftrightarrow \phi = \mathcal{M}(P \rightarrow T)$$

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

**Fig. 2.** The Matching Algorithm

*Proof.* We prove it by induction on the structure of pattern.

For the first case of the matching algorithm $\mathcal{M}$, we calculate as follows

$$
\begin{aligned}
&\ \ \phi = \mathcal{M}(\lambda x_1 \cdots x_l.\, P_1 \to \lambda x_1 \cdots x_o.\, T_1)\\
&\Leftrightarrow \quad \{\ \eta\text{-conversion}\ \}\\
&\ \ \phi = \mathcal{M}(\lambda x_1 \cdots x_l.\, P_1 \to \lambda x_1 \cdots x_l.\, T_1\ x_{o+1} \cdots x_l)\\
&\Leftrightarrow \quad \{\ \text{induction hypothesis}\ \}\\
&\ \ \phi \vdash \lambda x_1 \cdots x_l.\, P_1 \to \lambda x_1 \cdots x_l.\, T_1\ x_{o+1} \cdots x_l\\
&\Leftrightarrow \quad \{\ \eta\text{-conversion}\ \}\\
&\ \ \phi \vdash \lambda x_1 \cdots x_l.\, P_1 \to \lambda x_1 \cdots x_o.\, T_1
\end{aligned}
$$

For the second case, we assume

$$\phi = \phi_1. \cdots .\phi_m$$

and we derive

$$
\begin{aligned}
&\ \ \exists \phi.\, \phi = \mathcal{M}(\lambda \bar{x}.\, c\ E_1\ \cdots\ E_m \to \lambda \bar{x}.\, d\ T_1\ \cdots\ T_m)\ \wedge\ c = d\\
&\Leftrightarrow \quad \{\ \text{definition of } \mathcal{M}\ \}\\
&\ \ \exists \phi.\, \phi = \mathcal{M}(\lambda \bar{x}.\, E_1 \to \lambda \bar{x}.\, T_1). \cdots .\mathcal{M}(\lambda \bar{x}.\, E_m \to \lambda \bar{x}.\, T_m)\\
&\Leftrightarrow \quad \{\ \text{assumption}\ \}\\
&\ \ \exists \phi_1, \ldots, \phi_m.\, \phi_1 = \mathcal{M}(\lambda \bar{x}.\, E_1 \to \lambda \bar{x}.\, T_1), \ldots, \phi_m = \mathcal{M}(\lambda \bar{x}.\, E_m \to \lambda \bar{x}.\, T_m)\\
&\Leftrightarrow \quad \{\ \text{induction hypothesis}\ \}\\
&\ \ \exists \phi_1, \ldots, \phi_m.\, \phi_1 \vdash \lambda \bar{x}.\, E_1 \to \lambda \bar{x}.\, T_1, \ldots, \phi_m \vdash \lambda \bar{x}.\, E_m \to \lambda \bar{x}.\, T_m\\
&\Leftrightarrow \quad \{\ \text{definition of } \vdash\ \}\\
&\ \ \exists \phi_1, \ldots, \phi_m.\, \phi_1(\lambda \bar{x}.\, E_1) =_{\alpha\beta\eta} \lambda \bar{x}.\, T_1, \ldots, \phi_m(\lambda \bar{x}.\, E_m) =_{\alpha\beta\eta} \lambda \bar{x}.\, T_m\\
&\Leftrightarrow \quad \{\ \text{property of } (.)\ \}\\
&\ \ \exists \phi.\, \phi\ (\lambda \bar{x}.\, c\ E_1 \cdots E_m) =_{\alpha\beta\eta} \lambda \bar{x}.\, d\ T_1 \cdots T_m\\
&\Leftrightarrow \quad \{\ \text{assumption}\ \}\\
&\ \ \exists \phi.\, \phi \vdash \lambda \bar{x}.\, c\ E_1\ \cdots\ E_m \to \lambda \bar{x}.\, d\ T_1\ \cdots\ T_m\ \wedge\ c = d
\end{aligned}
$$

as required. Since the third case is similar to the second case, we omit the proof.

Since the fourth case is rather complex, we prove sufficient and necessity conditions separately.

($\Leftarrow$) For the case, a rule is $\lambda x_1 \cdots x_l.\, p\ E_1\ \cdots\ E_m \to \lambda x_1 \cdots x_l.\, T_1$. Let

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

Since *discharge* satisfies the property

$$(\lambda y_1 \cdots y_m.\, B)\ E_1 \cdots E_m = T_1$$

the following matching property holds.

$$\{p \to \lambda y_1 \cdots y_m.\, B\} \vdash \lambda x_1 \cdots x_l.\, p\ E_1\ \cdots\ E_m \to \lambda x_1 \cdots x_l.\, T_1$$

($\Rightarrow$) To obtain $\phi \vdash \lambda x_1 \cdots x_l\,.\, p\ E_1\ \cdots\ E_m \to \lambda x_1 \cdots x_l\,.\, T_1$, all the free variables in $T_1$ must be discharged. For each $E_i$ and each occurrence $E_i$ in $T_1$, we can choose whether we discharge it or not. By the definition of $\mathcal{DSP}$, $\forall i, j (i \neq j)\,.\, E_i \ntrianglelefteq E_j$ holds. Therefore if we don't discharge it, some free variables in $T_1$ will remain, resulting in no match. Thus we must discharge all the occurrences of $E_i$ in $T$. This operation matches *discharge* $[(y_1, E_1), \ldots, (y_m, E_m)]\ T_1$. □

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

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

**Theorem 5 (Efficiency).** Let $P$ be $\mathcal{DSP}$, $n$ be the size of term $T$, and $m$ be the maximum number of arguments of free variables in $P$. The algorithm $\mathcal{M}(P \to T)$ has the time complexity of $\mathrm{O}(mn)$.

*Proof.* Except for the second last case, it is straightforward that the time complexity of $\mathcal{M}$ is in proportion to the size of the pattern. For the last case, the function *discharge* traverses the term and the function *replace* checks for each argument $E_i$. *replace* costs $\mathrm{O}(m)$ and therefore *discharge* costs $\mathrm{O}(mn)$. □

It is worth noting that $m$ is generally quite small and bound in practice, so our algorithm is nearly linear with respect to the program to be transformed.

We have implemented this matching algorithm in Template Haskell [15], and the experiments show that our matching algorithm is very fast. We have also tested many transformation examples used in the MAG system [3], and it is rather encouraging that almost all transformation rules (except for those using the third-order patterns) can be concisely described in terms of our deterministic second-order patterns.

## 5 Application to Program Transformation

$\mathcal{DSP}$ covers a wider class of patterns then Miller's higher-order patterns as you have seen in fusion transformation in Sect. 1 and Sect. 2. In this section, we show how our matching algorithm can be useful for mechanise the tupling transformation.

Tupling is a program transformation where several results are returned from a single traversal of a data structure. For example, the function computing an average of a list

```
average1 xs = sum xs / length xs
```

is transformed into

```
average2 xs = s/l
  where (s,l) = sumLength xs
```

where `s` stores the summation and `l` stores the length of the list `xs`.

The definition of `sumLength` is defined as

```
sumLength [] = (0, 0)
sumLength (x:xs) = (x + s, 1 + l)
  where (s,l) = sumLength xs
```

which is derived as follows. The base case is trivial.

```
  sumLength []
= { spec. of sumLength }
  (sum [], length [])
= { unfolding sum and length }
  (0, 0)
```

The recursive case of `sumLength` is derived by

```
  sumLength (x:xs)
= { spec. of sumLength }
  (sum (x:xs), length (x:xs))
= { unfolding sum and length }
  (x + sum xs, 1 + length xs)
= { introducing a new function p }
  p x (sum xs) (length xs)
= { introducing a new function p'
    s.t. p' = \x (y,z) -> p x y z }
  p' x (sumLength xs)
```

In fact, finding `p` is the matching problem

$$\lambda x\, xs.\, p\ x\ (sum\ xs)\ (length\ xs) \rightarrow (x + sum\ xs, 1 + length\ xs)$$

which is resolved by our matching algorithm and the following substitution can be automatically obtained.

$$\{p \mapsto \lambda y_1\, y_2\, y_3.\, (y_1 + y_2, 1 + y_3)\}$$

Substituting it into the definition of `p'`, we obtain the definition of `sumLength` as

```
sumLength (x:xs)
  = (\y1 (y2,y3) -> (y1+y2,1+y3)) x (sumLength xs)
```

As another example, to avoid repeated evaluation where a function generates several identical calls to itself, the function fib defined as

```
fib 0     = 1
fib 1     = 1
fib (n+2) = fib (n+1) + fib n
```

is transformed into

```
fastfib n = v where (_,v) = fibt n
fibt 0 = (1,1)
fibt (n+1) = (u+v,u) where (u,v) = fibt n
```

The derivation is reduced into the matching problem of the rule

$$\lambda n.\, p\ (\mathit{fib}\ n)\ (\mathit{fib}\ (n+1)) \to \lambda n.\, (\mathit{fib}\ (n+1) + \mathit{fib}\ n, \mathit{fib}\ (n+1))$$

which is resolved with the match

$$\{p \mapsto \lambda x\, y.\, (y + x, y)\}$$

## 6 Conclusion

We have proposed a class of *deterministic second-order patterns* $\mathcal{DSP}$ which has at most a single second-order match. Our matching algorithm is linear if we fix patterns. Our pattern is a simple and natural extension of Miller's higher-order pattern [10] which has single most general unifier.

It is a classical approach to restrict the form of patterns to generate desirable higher-order matching. For example, under some restriction, possibly undecidable fifth-order matching is decidable [9], infinite third-order matches are finite [3], and nondeterministic second-order matching is deterministic [10]. Second-order matching is NP-complete [8], whose efficient algorithm is proposed by Curien *et al.* [16]. But restricting the form of patterns, there are known to be fast matching algorithm [17]. For example, if any function variable appears at most once in a pattern, it is linear.

Pattern matching plays an important role in program transformations. Many systems use first-order patterns [18–20] and most of them use first-order matching [15, 21–23]. Exceptions are MAG [24] and KORSO [2]; they use higher-order matching, which enables concise specification of program transformation, and make transformation more abstract and more reusable. One of the matching algorithms used in MAG is one-step matching [3, 25] which returns at least complete second-order match for arbitrary second-order patterns; our matching algorithm covers only a restricted class of second-order patterns $\mathcal{DSP}$.

There is a trade-off between completeness and efficiency of matching algorithm. Since the type of patterns are statically distinguished in program transformation to some extent, we might want to choose algorithm for each patterns. Our prototype implementation of the deterministic matching algorithm, together with the experiments on using it to code the examples tested in the MAG system, shows that our approach is promising. Actually, our patterns cover 91% of patterns in transformation rules in example transformations of MAG(ver.2.1) while Miller's higher-order patterns cover 71% and first-order patterns cover 69%.

In this paper, we focus on the second-order patterns. We believe the technique is applicable to higher-order patterns. We are now working on this.

## References

1. Huet, G., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Informatica **11** (1978) 31–55
2. Bruckner, B., Liu, J., Shi, H., Wolff, B.: Towards correct, efficient and reusable transformational developments. In: KORSO: Methods, Languages, and Tools for Construction of Correct Software. Volume 1009 of LNCS., Springer-Verlag (1995) 270–184
3. de Moor, O., Sittampalam, G.: Higher-order matching for program transformation. Theoretical Computer Science **269** (2001) 135–162
4. Gill, A., Launchbury, J., Jones, S.L.P.: A short cut to deforestation. In: Proceedings of the 6th International Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark, ACM Press (1993) 223–232
5. Sheard, T., Launchbury, J.: Warm fusion: Deriving build-catas from recursive definitions. In: Conference on Functional Programming Languages and Computer Architecture, ACM (1995) 314–323
6. Bird, R.: Introduction to Functional Programming using Haskell (second edition). Prentice Hall (1998)
7. Johann, P., Visser, E.: Warm fusion in stratego: A case study in the generation of program transformation systems. Technical Report Technical Report UU-CS-2000-43, Institute of Information and Computing Sciences, Utrecht University (2000)
8. Baxter, L.: The complexity of unification. PhD thesis, Department of Computer Science, University of Waterloo (1977)
9. Schubert, A.: Linear interpolation for the higher order matching problem. In: Theory and Practice of Software Development. (1996) 441–452
10. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation **1** (1991) 497–536

11. Nipkow, T.: Functional unification of higher-order patterns. In: 8th IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press (1993) 64–74
12. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Proceedings of the 5th International Conference on Functional Programming Languages and Computer Architecture (FPCA'91). Volume 523 of LNCS., Cambridge, Massachusetts, Springer-Verlag (1991) 124–144
13. Sheard, T., Fegaras, L.: A fold for all seasons. In: Conference on Functional Programming Languages and Computer Architecture. (1993) 233–242
14. Heckmann, R.: A functional langauge for the specification of complex tree transformation. In: Proc. ESOP. Volume 300 of LNCS. (1988) 175–190
15. Sheard, T., Peyton Jones, S.L.: Template metaprogramming for haskell. In: Haskell Workshop. (2002)
16. Curien, R., Qian, Z., Shi, H.: Efficient second-order matching. In Ganzinger, H., ed.: Proceedings of the 7th International Conference on Rewriting Techniques and Applications, New Brunswick, New Jersey, Springer-Verlag LNCS 1103 (1996) 317–331
17. Hirata, K., Yamada, K., Harao, M.: Tractable and intractable second-order matching problems. In: Computing and Combinatorics, 5th Annual International Conference, COCOON '99. Volume 1627., Tokyo, Japan (1999) 432–441
18. Guttmann, W., Partsch, H., Schulte, W., Vullinghs, T.: Tool support for the interactive derivation of formally correct functional programs. In: Journal of Universal Computer Science. Volume 9. (2003) 173–188
19. Peyton Jones, S.L., Tolmach, A., Hoare, T.: Playing by the rules: rewriting as a practical optimisation technique in ghc. In: Haskell Workshop. (2001)
20. Visser, E.: Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. In Middeldorp, A., ed.: Rewriting Techniques and Applications (RTA'01). Volume 2051 of LNCS., Springer-Verlag (2001) 357–362
21. Bauer, F., Ehler, H., Horsch, B., Möller, B., Partsch, H., Paukner, O., Pepper, P., eds.: The Munich Project CIP—Volume II: The Program Transformation System CIP-S. Volume 292 of LNCS. Springer-Verlag (1987)
22. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proc. Conference on Partial Evaluation and Program Manipulation, Amsterdam, ACM Press (1997) 203–217
23. Tullsen, M., Hudak, P.: An intermediate meta-language for program transformation. Research report yaleu/dcs/rr-1154, Department of Computer Science, Yale University (1998)
24. de Moor, O., Sittampalam, G.: Generic program transformation. In: Proc. of the 3rd International Summer School on Advanced Functional Programming. Volume 1608 of LNCS., Braga, Portugal, Springer-Verlag (1998) 116–149
25. Sittampalam, G.: Higher-order matching for program transformation. PhD thesis, University of Oxford (2001)