Design and Implementation of Deterministic Higher-order Patterns

Tetsuo Yokoyama[1], Zhenjiang Hu[1], and Masato Takeichi[1]

Department of Mathematical Informatics,
Graduate School of Information Science and Technology,
The University of Tokyo
{tetsuo_yokoyama,hu,takeichi}@mist.i.u-tokyo.ac.jp

Abstract. We introduce a class of deterministic higher-order patterns to Template Haskell for supporting declarative transformational programming with more elegant binding of pattern variables. Higher-order patterns are capable of checking and binding subtrees far from the root, which is useful for program manipulation. However, there are three major problems. First, it is difficult to explain why a particular desired matching result cannot be obtained because of the complicated higher-order matching algorithm. Second, the general higher-order matching algorithm is of high cost, which may be exponential time at worst. Third, the (possibly infinite) nondeterministic solutions of higher-order matching prevents it from being used in a functional setting. To resolve these problems, we impose reasonable restrictions on higher-order patterns to gain predictability, efficiency and determinism. We show that our deterministic higher-order patterns are powerful to support concise specification and efficient implementation of various kinds of program transformations for optimizations.

1 Introduction

In Glasgow Haskell Compiler (GHC), programmers can write rewrite rules as a part of the source program (in a pragma), suggesting the compiler to optimize the code wherever it can apply these rules [21]. Here is an example.

```
map f []     = []
map f (x:xs) = f x : map f xs
{-# RULES
  "map/map" forall f g xs.  map f (map g xs) = map (f.g) xs
 #-}
```

The first two lines define the familiar map function. The RULES pragma surrounded by `{-# RULES ... #-}` specifies a rewrite rule named "`map/map`". In this rule, RHS is more efficient than LHS because the intermediate list passed between the two `map` functions is eliminated. Althogh this kind of optimization knowledge is beyond automatic synthesis of the compiler, the programmer can instruct the compiler how to improve the code using the rule without the need to deform the map definition or to put hand into the core of the compiler.

The implementation of the RULES pragma is cheap. It adopts simple patterns in the rule specification, uses a trivial rule application strategy on the program, disallows side conditions, and therefore needs little extra compilation time. It has been shown [21, 13, 27] that such RULES pragmas can be successfully applied to implement optimizations from simple rules like "`map/map`" to complex rules like partial evaluation or the shortcut fusion.

Being simple, the RULES pragma is not as powerful as we may expect. Consider the *foldr promotion rule*, a known calculation rule [2] to push a functiuon f into a *foldr*.

$$\frac{f\ (g\ x\ y) = g'\ x\ (f\ y)}{f \circ \mathit{foldr}\ g\ z = \mathit{foldr}\ g'\ (f\ z)}$$

We may wish to write the following RULE pragma to instruct the GHC to optimize programs.

```
{-# RULES
     "foldr_promotion"  forall f g z.
          f . foldr g z = let g' x (f y) = f (g x y)
                          in foldr g' (f z)
 #-}
```

with the hope that GHC can automatically optimize the program like

```
sum . foldr (\x y -> double x : y) []
```

to

```
foldr (\x y -> double x + y) 0.
```

Unfortunately, this description is not acceptable by current GHC, due to the use of higher-order patterns. The pattern `g' x (f y)` is not a simple constructor pattern in the sense that the pattern variable `g'` appears in the function position and will be bound to a function value.

In this paper, we introduce a class of *higher-order patterns* to Template Haskell [23] for supporting declarative transformational programming with more elegant binding of pattern variables. Though the general higher-order patterns are capable of checking and binding subtrees far from the root, which is useful for program manipulation, the exponential time complexity and (possibly infinite) nondeterministic solutions of higher-order matching prevent them from being used in a functional setting.

We resolve these problems by imposing restrictions on the higher-order patterns, and a suitable order on the solutions of higher-order matching. As a result, our matching algorithm is deterministic and in linear time to the size of a given terms. We show that our higher-order patterns with meta-programming mechanism is powerful enough to enable concise specification and efficient implementation of various kinds of program optimizations. The main contributions of this paper can be summarized as follows.

– We propose a deterministic higher-order matching algorithm, which can efficiently compute out the largest solution with respect to an order on solutions. This solves the technical problems of higher-order matching, which is complex and nondeterministic.
– We introduce mechanism of higher-order patterns and higher-order matching into Template Haskell, by implementing a monadic combinator library. As a result, programmers can utilize the expressive first-class patterns and meta programming to specify abstract and reusable calculations.
– To show practical usefulness of our ideas, we have implemented many interesting program calculations such as fusion and tupling using our library, all of which are available on our web page[1]. The calculation rules are constructed in the compositional way, in which our basic combinators play an important role. All the code can be efficiently run on GHC 6.4.

The organization of this paper is as follows. In Section 2, we introduce higher-order patterns, define an order on the matches, and show that our matching produces at most a single match. In Section 3, we implement our idea as a combinator library which enhances the expressive power of Template Haskell with first class higher-order patterns. Section 4 discusses the related works, and Section 5 concludes the paper.

2 Higher-order Patterns

Three limitations exist in the ordinary constructor-based patterns (hereafter we call them simple patterns) for programming transformations, which we want to overcome. (1) The simple patterns only check the region very near to the root and bind subtrees rooted at the region to pattern variables. (2) The simple patterns do not allow local bindings, making them difficult to capture the scope (local binding) of variables or the template of programs. (3) The simple patterns are not first-class, in the sense that they cannot be named, passed as parameters, or constructed from the smaller components.

To overcome the first limitation so that the programmer can specify a region arbitrarily far from the root satisfying some condition [9, 10], we extend the simple patterns with function variables, i.e., higher-order patterns. Recall the higher-order pattern `g' x (f y)` in the introduction where `g'` is a function variable. This higher-order pattern can be used to be matched with an expression where `x` and `f y` appear at any place, possibly very far away from the root of the given expression. For the second, we allow lambda abstraction to appear in patterns. For instance, the pattern $f\ (\lambda x.\, x + 1)$, where f is a function variable, can be used to extract an expression template in the form of "something plus 1" from a given expression. For the third, we may utilize meta programming mechanism to define a framework with which patterns can be created. We will focus on the solution to the first two limitations in this section, and leave the solution to the last limitation later.

[1] `http://www.ipl.t.u-tokyo.ac.jp/yicho/`

2.1 Expressions and Higher-order Patterns

To be concrete, we explain our idea with the simplest form of expression: the expressions that are built recursively from variables, data constructor/constants, abstraction, or application.

$$exp ::= var \mid con \mid \lambda v.\ exp \mid exp\ exp$$

An η-redex is one of the form $\lambda x.E\ x$ where E does not contain x. An expression is called η-normal if it contains no η-redex. An expression is called $\beta\eta$-normal or simply *normal* if it contains no β-redex and η-redex. We shall generally ignore problems of collision of bound variables. Given expression E, a set of free variables in E is written as $FV(E)$. In this paper, we will write v, w, x, y, z for variables, f, g for higher-order variables, a, b, c for constants, B, E, F, P, T for general expressions, ϕ for substitutions, and i, j, k, l for indexes.

Different from the simple patterns which are of a very restricted form, our higher-order patterns (or called patterns if this is clear from the context) basically cover all expressions.

$$hpat ::= var \mid con \mid \lambda v.\ hpat \mid hpat\ hpat$$

Different from the simple patterns, the higher-order patterns allow occurrences of free variables in the function position, lambda abstraction, and non-saturated constructors. Formally, the *order* of a type is recursively defined as

$$\begin{aligned} &order\ \text{GroundType} = 1 \\ &order\ (T_1 \to T_2) \quad = max\ (order\ T_1 + 1)\ (order\ T_2) \end{aligned}$$

The order of an expression is the order of its type. The order of a pattern is the maximum order of free variables occurring in it. The pattern is called *higher-order* if the order is more than one.

Generally, the higher-order matching problem is at least NP-hard [1]. We impose restrictions on the higher-order patterns to obtain matches in linear time. Before explaining the restrictions, we prepare the necessary definitions.

The *head* of an expression is the expression itself if it is a variable or a constant, the head of the functional part if it is an application, and the head of the body if it is a lambda abstraction. A pattern is *flexible* if its head is a free variable, and *rigid* otherwise. For example, patterns x and $f\ a\ 1$ are flexible where the latter is a flexible higher-order pattern. But $x + x$ and $\lambda y.\ y + x$ are rigid. A pattern is *linear* if no repeated free variables and the formal parameters of the lambda abstraction of the arguments of the free variables occur in the same pattern. For examples, $x + 1$ and y are linear, but $x + x$ is not. Neither does $f\ (\lambda x.\ x+x)$. A pattern is *flat* if no higher-order free variables of the pattern has free higher-order variables again in its arguments. For example, $f\ (1 + x)$ is flat, but $f\ (1 + g\ 2)$ is not because the higher-order variable g appears in the argument of the higher-order variable f.

A substitution is a mapping from variables to expressions, which is denoted as, for example, $\{f \mapsto \lambda x.\ x\ a\ b\}$.

We define equality ($==_\Box$) on the terms that may contain the hole $\Box$.

$$\begin{array}{ll} c ==_\Box c & = True \\ F\ E ==_\Box T_1\ T_2 & = (F ==_\Box T_1) \wedge (E ==_\Box T_2) \\ \lambda x.\ E ==_\Box \lambda x.\ T & = E ==_\Box T \\ \_ ==_\Box \Box & = True \\ \Box ==_\Box \_ & = True \\ x ==_\Box x & = True \\ \_ ==_\Box \_ & = False \end{array}$$

Here, α-renaming is implicitly assumed. This equality is the same as usual except that special constant $\Box$ is regarded as equal to any term.

We say that a term $E_1 (\neq \Box)$ is a *subterm* of E_2 (ignoring $\Box$'s), denoted by $E_1 \trianglelefteq_\Box E_2$, if $E_1 ==_\Box E_2'$ where $E_2' \in subTerm\ E_2$ and $subTerm$ is defined below:

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

2.2 Deterministic Higher-order Matching

Generally, higher-order matching, i.e., matching a higher-order pattern with an expression (term), is not deterministic and returns many and possibly infinite solutions (called substitutions or matches in this paper). However, some patterns may return at most a single match with any terms. We will specify a class of deterministic higher-order patterns. For program transformation, we may still want to use patterns that are not deterministic. For such patterns, for simplicity and predictability, we impose an order on solutions and picking up the largest as the result to gain determinism and efficiency.

We formally define the matching problem. A substitution is *normal*, if all expressions in its range are normal. A substitution is *closed*, if all expressions in its range are closed. A normal closed substitution ϕ is a *match* for the *matching pair* $P \to T$ with local variables $\Delta = \{x_1, \ldots, x_m\}$ where P and T are normal, if

$$\phi\ (\lambda x_1 \ldots x_m.\ P) =_{\alpha\beta\eta} \lambda x_1 \ldots x_m.\ T$$

We write this as

$$\phi, \Delta \vdash P \to T.$$

We call computing ϕ from $P \to T$ the matching problem. We write $\phi, \{\} \vdash P \to T$ as $\phi \vdash P \to T$.

The reason why we only consider the closed match is to prevent unbound variables appeared in the expression after applying a substitution. If the match has unbound variables, the variables can be instantiated into any expressions. This nondeterminism does not suit the functional setting.

In the rest of this section, we firstly introduce a class of deterministic patterns which induce a linear time matching algorithm with respect to the size of input terms. Then, for the other patterns, we restrict matching algorithm to return at most a single match and the matching algorithm becomes linear.

Deterministic Patterns
First, we introduce a class of higher-order patterns and prove its determinism.

Definition 1 ($\mathcal{DHP}_3$). *The class of patterns $\mathcal{DHP}_3$ consists of at most third order linear patterns P in which the arguments $E_1, \ldots, E_m$ of any free variable occurring in a normalized term of P satisfy the following conditions. Let $\lambda z_i^1 \cdots z_i^{k_i}.\ B_i$ be E_i where B_i have no outmost lambda abstraction.*

(i) $\forall i.\, FV(E_i) \neq \{\,\}$
(ii) $\forall i, j.\ \{z_i^1 \mapsto \Box, \ldots, z_i^{k_i} \mapsto \Box\}\ B_i \not\trianglelefteq_{\Box} \{z_j^1 \mapsto \Box, \ldots, z_j^{k_j} \mapsto \Box\}\ B_j$
(iii) $\forall i.\, (v \in FV(E_i) \Rightarrow v \notin FV(P))$
(iv) *For all i, in B_i, variables $z_i^1, \ldots, z_i^{k_i}$ must appear just once* $\Box$

Free variables in the pattern of $\mathcal{DHP}_3$ should have the order of at most 3, therefore variables $z_i^1, \ldots, z_i^{k_i}$ in B_i must be first-order. (i) E_i should not be a closed term, so the term $f\ 1$ is not a valid pattern because the argument 1 of the free variable f is a closed term, which does not contain any free variable; (ii) For all $i, j (i \neq j)$, B_i is not an instance of a subterm of B_j, so $\lambda x.\ f\ (x{+}1)\ (\lambda y.\ x{+}y)$ is not a valid pattern since $x+1$ is an instance of $x+y$; (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 $f\ x$ is not a valid pattern; (iv) for example, $\lambda x.\ f\ (x+x)$ is valid, but $\lambda x.\ f\ (\lambda y.\ x\ y\ y)$ is invalid. For example, a pattern $\lambda x\ y.\ f\ x\ (\lambda z.\ c\ (y\ z))$ belongs to $\mathcal{DHP}_3$.

Lemma 1. *If P is a $\mathcal{DHP}_3$, there is at most a single match ϕ such that $\phi \vdash P \to T$.*

Proof. See Appendix. $\Box$

We say that a pattern is deterministic if given any term the matching with the pattern results in at most a single solution. The patterns in $\mathcal{DHP}_3$ are deterministic.

The inference rules of judgment of matches are described in Fig. 1. Here, $\phi_1 \uplus \phi_2$ returns the composition of ϕ_1 and ϕ_2. If the one of ϕ_1 and ϕ_2 is fail, then it returns fail. An expression $\phi \ominus xs$ restricts the domain of ϕ, which returns the substation which does not change the variables in xs and applies ϕ for the other variables. Almost all rules are trivial. The involved part is where the head of the pattern is flexible. If the pattern is $\mathcal{DHP}_3$, the linear-time matching algorithm returns the deterministic solution.

In FLEX1 rule, function *dreplaces* replaces all the instance of B_i by $v_i\ x_i^1\ \cdots\ x_i^{k_i}$ in which $x_i^1, \ldots, x_i^{k_i}$ are instantiated into corresponding subexpressions in T. For example, *dreplaces* $[(x+y, v\ x)]\ ((1+y)*(2+y))$ returns $v\ 1 * v\ 2$. Here, the instances of $x+y$ are $1+y$ and $2+y$, the corresponding subexpressions are $v\ 1$ and

$$\frac{FV(E) = \emptyset}{\{x \mapsto E\}, \{\} \vdash x \to E}\text{VAR} \quad \frac{}{\{\}, \{x\} \vdash x \to x}\text{BOUND} \quad \frac{}{\{\}, \{\} \vdash c \to c}\text{CON}$$

$$\frac{\phi, \Delta \vdash P \to T}{\phi, \Delta - \{x\} \vdash \lambda x.\ P \to \lambda x.\ T}\text{LAM1} \quad \frac{\phi, \Delta \vdash P \to T\ x}{\phi, \Delta - \{x\} \vdash \lambda x.\ P \to T}\text{LAM2}$$

$$\frac{\phi_1, \Delta_1 \vdash E_1 \to T_1 \quad \cdots \quad \phi_m, \Delta_m \vdash E_m \to T_m}{\phi_1 \uplus \cdots \uplus \phi_m, \Delta_1 \cup \cdots \cup \Delta_m \vdash c\ E_1 \cdots E_m \to c\ T_1\ \cdots\ T_m}\text{RIGID1}$$

$$\frac{\phi_1, \Delta_1 \vdash E_1 \to T_1 \quad \cdots \quad \phi_m, \Delta_m \vdash E_m \to T_m}{\phi_1 \uplus \cdots \uplus \phi_m, \{x\} \cup \Delta_1 \cup \cdots \cup \Delta_m \vdash x\ E_1 \cdots E_m \to x\ T_1\ \cdots\ T_m}\text{RIGID2}$$

$$\frac{\begin{array}{c}\lambda y_1 \cdots y_l.\ f\ (\lambda x_1^1 \cdots x_1^{k_1}.\ B_1) \cdots (\lambda x_m^1 \cdots x_m^{k_m}.\ B_m) \in \mathcal{DHP}_3 \\ FV(T') - \{v_1, \ldots, v_m\} = \emptyset \\ T' = \mathit{dreplaces}\ [(B_1, v_1\ x_1^1 \cdots x_1^{k_1}), \ldots, (B_m, v_m\ x_m^1 \cdots x_m^{k_m})]\ T\end{array}}{\{f \mapsto \lambda v_1 \cdots v_m.\ T'\}, FV(T) \vdash f\ (\lambda x_1^1 \cdots x_1^{k_1}.\ B_1) \cdots (\lambda x_m^1 \cdots x_m^{k_m}.\ B_m) \to T}\text{FLEX1}$$

$$\frac{\begin{array}{c}f\ (\lambda x_1 \cdots x_m.\ B) \in \mathcal{A}_{LT} \\ (T', \phi) = \mathit{replaceLT}\ (T, (B, v\ x_1 \cdots x_m)) \quad v \in FV(T')\end{array}}{\{f \mapsto \lambda v.\ T'\} \uplus (\phi \ominus \{x_1, \ldots, x_m\}), FV(T) \vdash f\ (\lambda x_1 \cdots x_m.\ B) \to T}\text{FLEX2}$$

Fig. 1. Matching Tree

v 2 respectively. Since the result match must be closed, $FV(\lambda v_1 \cdots v_m.\ T') = \emptyset$, i.e., $FV(T') - \{v_1, \ldots, v_m\} = \emptyset$.

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

By definition 1, B_i must not be a constant. Therefore, constant c is unchanged. For variables, we try to replace it by function *replace*. For lambda abstractions, firstly we try to replace it. If it fails, its body is recursively tried to be replaced. Otherwise, it is returned. Similarly, for applications, we try to replace it. If it fails, its function part and argument part are recursively tried to be replaced. Otherwise, it is returned.

Function *replace* takes a pair of list. The pair consists a term to be replaced and a term to be replaced with. T is checked to be an instance of B by syntactical matching *smatch*. If it is not, the next pair is checked. Otherwise, a term is

replaced with instantiated X by the match.

$$
\begin{array}{l}
\mathit{replace}\ [\,]\ T\ =\ T \\
\mathit{replace}\ ((B, X) : s)\ T\ = \\
\quad \textbf{let}\ \phi = \mathit{smatch}\ B\ T \\
\quad \textbf{in if}\ \phi = \mathit{fail}\ \textbf{then}\ \mathit{replace}\ s\ T\ \textbf{else}\ \phi\ X
\end{array}
$$

The following lemma describes the correctness of the matching where the patterns are $\mathcal{DHP}_3$.

Lemma 2. *If P is a $\mathcal{DHP}_3$ and there is a match ϕ such that $\phi \vdash P \to T$, matching tree in Fig. 1 returns ϕ.*

Proof. See Appendix. □

Restricting Matching Algorithm to be Deterministic
Sometimes, we need patterns other than deterministic ones for the expressive calculation rules. If the pattern is not deterministic, there might exist more than a single solution. In this case, we select a solution by means of the LT order. For simplicity and efficient matching algorithm, we impose restrictions on patterns: the patterns are linear, flat, upto third-order, the number of the arguments of free pattern variables being at most one. We call the patterns a class $\mathcal{A}_{LT}$. Our matching algorithm for such patterns depends on the following order on matches (solutions).

Definition 2 (LT Order). Left-to-right, top-down abstraction order on normal expressions according to free variable x is defined as follows

$$
\begin{array}{ll}
E \leq_x F & = (E <_x F)\ \vee\ (E == F) \\
 & \\
c <_x c & = \mathit{False} \\
x <_x x & = \mathit{False} \\
\_ <_x x & = \mathit{True} \\
E_1\ E_2 <_x F_1\ F_2 & = (E_1 <_x F_1) \vee ((E_1 == F_1) \wedge (E_2 <_x F_2)) \\
\lambda y.\ E <_x \lambda y.\ F & = E <_x F
\end{array}
$$

Here $(==)$ is α-equality. □

Given two expressions, the occurrence positions of x are compared in the left-to-right and top-down order. For example, the following expressions are lined in an increasing order[2].

$$(1 + 1) + x <_x x + (1 + 1) <_x x\ (1 + 1)$$

The LT order is extended on matches with corresponding matching pairs. Matches are compared by the LT order on the body of the range expressions from the higher-order variables according to the formal parameter of enclosing variables.

[2] Note that $x + x$ is a syntax sugar of $(+)\ x\ x$.

The order to check expressions is according to the order of the corresponding domain of the match in the patterns from left-to-right and top-down. For example, according to matching $p\ (\lambda x.\ x + x)$ with $(1 + 1) + (1 + 1)$, the order of match is

$$\{\ p \mapsto \lambda x.\ (1 + 1) + x\ 1\ \} < \{\ p \mapsto \lambda x.\ x\ 1 + (1 + 1)\ \} < \{\ p \mapsto \lambda x.\ x\ (1 + 1)\ \}$$

When matching $f\ 1 + g\ 1$ with $(1 + 1) + (1 + 1)$, since f is left to g, we compare the range of f first,

$$\{f \mapsto \lambda x.\ 1 + x, g \mapsto \lambda x.\ x + 1\} < \{f \mapsto \lambda x.\ x + 1, g \mapsto \lambda x.\ 1 + x\}.$$

Though the LT order is partial order, if we compare the matches which are generated from the same patterns and terms, it becomes total order.

Intuitively, to solve the matching pair where the pattern is flexible, it is necessary to operate β-abstraction. For example, $f\ 1 \to 1 + 1$ is transformed into $f\ 1 \to (\lambda x.\ x + 1)\ 1$ and $f\ x \to \lambda y.\ y + x$ is into $f\ x \to (\lambda w\ y.\ y + w)\ x$. The number of solutions corresponds to the number of ways to operate β-abstraction. We consider the match with precisely one β-abstraction. To obtain the largest match by means of the LT order, we need to β-abstract in left-to-right and top-down manner, which is formally described as follows:

$$\begin{array}{lll}
\mathit{replaceLT}\ (E, (B, X)) = \textbf{let}\ \phi = \mathit{smatch}\ B\ E & & \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{in if}\ \phi = \mathit{fail}\ \textbf{then}\ \mathit{dr}\ E & & \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (\phi\ X, \phi) & & \\
\textbf{where} & & \\
\mathit{dr}\ x & = (x, \mathit{idS}) & \\
\mathit{dr}\ c & = (c, \mathit{idS}) & \\
\mathit{dr}\ (E_1\ E_2) & = \textbf{let}\ (E_1', \phi_1) = \mathit{replaceLT}\ (E_1, (B, X))) & \\
 & \qquad\quad (E_2', \phi_2) = \textbf{if}\ \phi_1 = \mathit{fail} & \\
 & \qquad\qquad\qquad\qquad\quad \textbf{then}\ \mathit{replaceLT}\ (E_2, (B, X)) & \\
 & \qquad\qquad\qquad\qquad\quad \textbf{else}\ (E_2, \phi_1) & \\
 & \quad \textbf{in}\ (E_1'\ E_2', \phi_2) & \\
\mathit{dr}\ (\lambda y_1 \cdots y_m.\ E) = \mathit{replaceLT}\ (E, (B, X)) & &
\end{array}$$

Function *replaceLT* takes a term E, a pattern B to be replaced and an expression X to be replaced with. Term X includes variables $x_1, \ldots, x_m$, which are instantiated eventually. Function *replaceLT* tries to replace B with X using usual syntactical matching *smatch*. If the matching succeeds, we use the match to instantiate X, terminate traverse, and return a pair of instantiated X and the match. Otherwise, we continue traversing by *dr*. Function *dr* is one layer traversing, which traverses the expression of the given term. Here, *idS* is the identity substitution. If the expression is application, we recursively try to do β-abstraction in the functional part first and then the argument part. If the expression is a lambda abstraction, we recursively try to do β-abstraction in the body.

For the pattern of $\mathcal{A}_{LT}$, if we map higher-order variables to the corresponding terms with the enclosing formal parameter, we can obtain the match. For

example, $\{f \to \lambda x.\ 1\} \vdash f\ 1 \to 1$. But, this match is not interesting, since we cannot replace anything by such match. We call the match *trivial* if the outmost formal parameters of one of the target expressions of the match does not appear at its body.

Lemma 3. *If P is a $\mathcal{A}_{LT}$ and there exists a nontrivial match ϕ such that $\phi \vdash P \to T$, matching tree in Fig. 1 will be guaranteed to return the maximum match of ϕ with respect to the LT order.*

Proof. See Appendix. □

Theorem 1. *If P is a $\mathcal{DHP}_3$ or a $\mathcal{A}_{LT}$, and there exists ϕ such that $\phi \vdash P \to T$, we can obtain one of match ϕ from P and T by matching tree in Fig. 1 in linear time.*

Proof. See Appendix. □

3 Yicho: A Combinator Library for Programming with Higher-order Patterns

To bring our higher-order patterns and efficient pattern matching algorithm into practical use, we have designed and implemented Yicho, a monadic combinator library for supporting declarative specification of program transformation in Template Haskell [23], a meta extension to Haskell 98. The combinator library uses higher-order patterns as first-class values which can be passed as parameters, constructed by smaller ones in compositional way, and returned as values. As a result, our library provides more flexible binding than simple ones, and enables more abstract and modular description of program transformation than the libraries Template Haskell provides.

3.1 Template Haskell

Before explaining Yicho, we briefly review the Template Haskell upon which Yicho is built. Template Haskell [23][3] provides a mechanism to handle abstract syntax trees of Haskell in Haskell itself, whose data type is defined as

```
data Exp = VarE Name | ConE Name | AppE Exp Exp | LamE [Pat] Exp |...
```

Enclosing quasi-quote brackets `[| |]` make concrete syntax of Haskell programs abstract syntax tree whose type is `ExpQ` (i.e., `Q Exp`), where `Q` monad encapsulates name manipulation, reification, failure, input/output, and so on. For example, for the function `sum` in the Haskell module `Data.List` for calculating the sum of a list, `[| sum |]` has type `ExpQ`.

Expressions are spliced at compile time by placing `'$'` before the expressions. For example, function `$([| sum |])` has the same type as `sum`, i.e., `[Int] -> Int`[4].

[3] The current Template Haskell has been changed a little from [23], with some new features [22].

[4] Strictly speaking `sum :: (Num a) => [a] -> a`.

3.2 An Example Use of Yicho

To get a flavor of Yicho, we show how to use Yicho to code program transformation, and how we can run it in Haskell. Recall the fold promotion rule in the introduction. It can be defined in Yicho as

```
promotion :: RuleY -> ExpQ -> Y ExpQ
promotion laws exp = do
  [f,g,z,z',g'] <- pvars ["f","k","z","z'","k'"]
  [| $f . foldr $g $z |] <=== laws exp
  [| \x xs -> $g' x ($f xs) |] <=== laws [| \x xs -> $f ($g x xs) |]
  ret [| foldr $g' ($f $z) |]
```

Function `promotion` takes calculation rule `RuleY` and code `ExpQ` and returns a code with its environment `Y ExpQ`. In the third line, `f`, `k`, `z`, `z'`, `k'` are declared to be variables; the unquote `$` is actually splicing the expression, but, intuitively we can regard expression `$x` as a meta variable of name `$x`. We deal with these variables rigorously in Section 3.3. In the fourth line, `exp` is transformed by the rule `laws` and the result is matched with the pattern `[| $f . foldr $g $z |]`. Implicitly, the mappings from the variables `$f`, `$g`, `$z` are bound in the monadic environment. The next two lines are a straightforward translation of the original promotion rule. `$f` and `$g` in the both hand side of `<===` are instantiated and matched and the result match is added to the environment. This pattern instantiation contributes the modularity of patterns. Here, the higher-order patterns such as

```
[| \x xs -> $g' x ($f xs) |]
```

play an important role in this concise specification. Finally, the result expression with its environment is returned by `ret`.

Provided that we have an unfolding rule `rule1` for `sum` and `double`, a pretty printing function `prettyExpQ :: ExpQ -> IO ()`, we define two expressions:

```
ex1 = [| sum . foldr (\x y -> double x : y) [] |]
ex2 = runY (promotion rule1 ex1)
```

where the `ex2` is obtained by applying promotion rule to `ex1`. We can check this by using pretty printing function

```
GHCi> prettyExpQ ex2
foldr (\x_1 -> (+) (2 * x_1)) 0
```

We can compare the efficiency of the two expressions as follow.

```
GHCi> $ex1 (take 100000 [1..])
10000100000                    (0.33 secs, 21243136 bytes)
GHCi> $ex2 (take 100000 [1..])
10000100000                    (0.27 secs, 19581216 bytes)
```

It is worth noting that the promotion theorem is applied in `ex2` at compile time, and the function `$ex2` is certainly improved both in the execution time and the heap size.

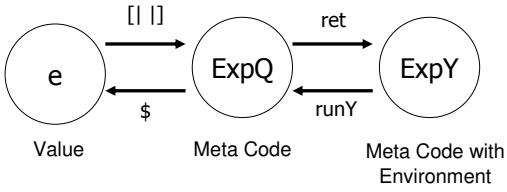

Fig. 2. Relationship of Types

3.3 Implementation Issues

ExpY: A Monad for Programming Transformation
During program transformation, we need to manipulate environments, deal with failures, and cope with renaming to avoid name conflicts. We capture these by defining a monad.

Monad is a way to structure programming and provides an easy treatment of program. One may construct a new monad, combining aspects of both operating lookup and update of environment and keeping track of a value or a failure of match. A cheaper way is to define a combined monad consisting of smaller ones.

```
type ExpY = Y ExpQ
type Y e = StateT Subst (ErrorT String Q) e
```

Type `Subst` represents an environment, a map from variables to closed expressions, and `StateT` and `ErrorT` are monad transformers which are defined in Haskell Hierarchical Libraries of GHC. Intuitively, state transformer `StateT` conveys states and `ErrorT` monad conveys a value or a failure of match.

The reason why we choose `Y ExpQ` instead of `Y Exp` is for the ease of manipulation of expressions of type `ExpQ`; using arrow (`<-`) of do notation we can easily take out expressions of type `ExpQ`, we can use quasi-quote brackets to make them, and Template Haskell provides a lot of useful functions on `ExpQ`.

We use function `ret` to lift type `ExpQ` into type `ExpY`. Conversely, we use function `runY` to bring back to type `ExpQ` from type `ExpY`. The relationship of those types is summarized in Figure 2.

Construction of Higher-order Patterns
In our library, patterns are expressions. Therefore, patterns are first-class. The simple variable patterns can be written as `[| $f |]`. So far, we regarded `$f` as a meta variable named `$f`. As we mentioned before, this is not really correct. Actually, unquote `$` splices the expression substituted into `f`. Therefore, variable `f` must be bound before. The code should look like

```
do f <- pvar "f"
   ... [| $f |] ...
```

Match	`(<==) :: ExpQ -> ExpQ -> Y ()`
Rule	`(==>) :: ExpQ -> ExpQ -> RuleY`
Choice	`(<+) :: Choice a => a -> a -> a`
Case Alternative	`caseM :: ExpQ -> [RuleY] -> ExpY`
Sequence	`(>>) :: ExpY -> ExpY -> ExpY`

Fig. 3. Basic Combinators

We declare `f` as a free variable appeared in patterns. Function `pvar` defined by

```
pvar :: String -> Y ExpQ
pvar s = liftY (do { n <- newName s
                   ; return (varE n) })
```

takes `String` and generates a new name `n`. This name is wrapped by `Q` monad twice. This is important since we use the outer `Q` monad for generating new names and the inner monad for representing code. Here, the function

```
liftY :: Q a -> Y a
liftY = lift . lift
```

is a monad transformer [14], where the first `lift` transforms `Q` monad into `ErrorT String Q` monad which is lifted into `Y` monad by the second `lift`. The outer `Q` is wrapped in `Y` monad. The both two `Q` monads are merged by `runY` when we move back to the usual world.

A list of pattern variables can be defined by function `pvars`, which takes a sequence of names `String` and returns a list of pattern variables:

```
pvars :: [String] -> Y [ExpQ]
pvars = sequence . map pvar
```

For example, the expression

```
do [oplus,e] <- pvars ["name1","name2"]
```

defines `oplus` and `e` as pattern variables whose names are `name1` and `name2` respectively.

Basic Combinators

Our library has five important combinators; match (`<==`), rule (`==>`), deterministic choice (`<+`), case-selection `caseM`, and sequence (`>>`). Types of the basic combinators are summarized in Fig. 3.

Among these combinators, the essential construct is match:

```
pat <== term
```

which yields a normal frozen closed substitution (match) that makes patterns (`pat`) and terms (`term`) to be equal. Before matching, both the pattern and the term are reduced into normal form, and free variables of both `pat` and `term` which are appeared in `term` are frozen. If there is no match, then the function `fail` is called and the rest of programs is not evaluated until the failure match is captured. If the match succeeds, the match, of which frozen variables in target expressions are thawed, is added to the environments in `Y`.

For example, matching

```
[| \x xs -> $oplus x (bign xs, sum xs) |] <==
     [| \x xs -> if x > sum xs then x : bign xs else bign xs |]
```

yields a substitution

```
{ $oplus := \x (b,s) -> if x > s then x : b else b }
```

Recall that annotation `$` means unquote. Thus, this match is equivalent to

```
{ oplus := [| \x (b,s) -> if x > s then x : b else b |] }
```

Function variable `$oplus` is second-order and to obtain the match we used higher-order matching described in Section 2.

The obtained match is kept tracked by the `Y` monad and it is applied in match combinator, and in calling function `ret`, which is `return` of `Y` monad.

The rest of combinators other than match can be constructed by the match combinator and/or functions in the standard library. Before explaining the rest of combinators, we introduce the type which represents the transformation rules.

A transformation rule is taking an expression and returns a lifted expression.

```
type RuleY = ExpQ -> ExpY
```

A transformation rule is constructed by operator `(==>)`.

```
(pat ==> body) term = do pat <== term
                         ret body
```

Operator (`<+`) is deterministic choice defined on `Choice` class.

```
class Choice a where
    (<+) :: a -> a -> a
```

If `a` is `ExpY`, it returns the first argument if it is not error. Otherwise, it returns the second argument. It is implemented by specializing function `mplus` in a standard library which is a member of `MonadPlus` monad.

```
instance Choice ExpY where
    (<+) = mplus
```

Choice operator (`<+`) can be applied to `RuleY` in the sense that it applies the first rule to the given expression and returns the result if it succeeds, otherwise the second rule is applied to the given expression.

```
instance Choice RuleY where
    (r1 <+ r2) x = r1 x `mplus` r2 x
```

Using this operator, a meta version of `case` can be defined by

```
caseM sel []     = fail "in caseM"
caseM sel (r:rs) = r sel <+ caseM sel rs
```

It attempts to apply to `sel` the rules of the second argument list from left to right, until it succeeds, and the first succeeded result is returned. If all the rules result in fail, the failure with the error message is returned.

For simplicity, we use longer arrows (`<===`) and (`===>`). They are the same as short arrows except that types are different.

```
(<===) :: ExpQ -> ExpY -> Y ()          -- match
(===>) :: ExpQ -> ExpY -> RuleY         -- rule
```

Sequencing of binding new environments can be realized by combining matches with operator (`>>`).

```
(pat1 <== term1) >> (pat2 <== term2)
```

which can be written as a sequence of match using do notation.

```
do pat1 <== term1
   pat2 <== term2
```

Using these important combinators, we can define many new combinators. For example, combinator `success` always returns the input value and combinator `try` applies a given rule if possible.

```
success x = ret [| $x |]
try r = r <+ success
```

For programmer, these new combinators seem not different from built-in combinators. This means extensibility of our library.

3.4 Other Examples

Many calculations can be efficiently implemented by our library. Interested readers are invited to visit Yicho web page. We give a list of some of them: (1)Promotion, (2)Tupling, (3)Warm Fusion, (4)Accumulation, (5)Parallelization, (6)Shortcut Fusion, (7)Maximum Segment Sum Problem, (8)Calculation of numerical expression, and so forth. They have all been tested on GHC 6.4.

4 Related Works

Mohnen introduces restricted higher-order patterns, so called context patterns [17] into Haskell, and in the sequence paper to remove unnecessary repeated traversing he introduces extended context [18]. Heckmann introduces a special constant '@' which means a kind of hole of the context, and thus the matching becomes nondeterministic [9]. This pattern is implemented in TrafoLa, which are divided into a deterministic one (D-TrafoLa) which selects one of the matches each time matching succeeds and a nondeterministic one (N-TrafoLa). Our approach is similar to D-TrafoLa.

Wadler proposes *views* that enables pattern matching on abstract data types by user supplied isomorphic mapping between them [28]. The descendant version of views are [3, 19]. Similar mechanism called *laws* allows pattern matching with non-free algebraic data types [24]. We take no account for this aspect in this paper.

Fahndrich and Boyland make a pattern like function with which one can define recursive patterns, alternative patterns, and so on [7]. Tullsen makes patterns functions of type `a->Maybe b` [25]. Therefore, patterns are first-class as well as functions. He specifies combinators for constructing patterns. He uses `Maybe` monad and its generalization `MonadPlus` monad to handle failures explicitly. Our pattern uses `ErrorT` monad and the failure can be propagated, and are the expression itself and thus are first-class objects. Some calculi introduces explicit failure constructs and capture of them [20, 11].

Domain-specific language in meta-programming is a promising approach [5]. Among functional meta-programming languages, i.e., Template Haskell [23], MetaML [15], and MetaOCaml [4, 16], only Template Haskell provides a way to construct and pattern match with abstract syntax trees. In the others, the code fragments can not contain unbound variables. This is matched with our aim; we want to change the semantics of patterns. Therefore, we introduce reconstruction of abstract syntax trees before passing it to compiler.

The supposed application of the combinator library we provide is program transformation. There are known to be many transformation systems. TrafoLa [9] and KORSO [12] are the functional transformation systems with kind of higher-order patterns. MAG [6] is higher-order term rewriting system for automatic application of promotion. Although they produce nondeterministic matches and the matching algorithm is not linear to the given term, higher-order patterns in MAG cover our patterns and many calculations can be written concisely. Stratego is a domain-specific language for writing term rewriting system for program transformation [26]. Almost all the program transformation systems adopt CUI. Notable exception includes Ultra, an interactive program transformation system which has visual interface [8].

5 Conclusion and Future Work

We define a class of deterministic higher-order patterns which cover a wide class of important patterns in program transformation. For those other than determin-

istic patterns, we restrict matching algorithm to be deterministic by imposing a suitable order, and propose an efficient linear-time matching algorithm. Although the matching algorithm may not be complete, meta programming can be used to recover the completeness.

We have implementd the higher-order patterns for Template Haskell and present a combinator library for declarative transformational programming. With our library, one can write calculation theorems almost as it is. We have seen that many calculation rules can be concisely described by our library.

Yicho can manipulate only a subset of the syntax of expression `Exp`. For example, the matching algorithm precludes let, case and do statements and records. It is future development to cover the more language construct of Haskell. We plan to abstract calculation strategies, and classify the calculation rules and modularize calculation functions. For example, we can define promotion functions on lists and trees, but we can abstract the common program structure and integrate them into a single generic function.

Acknowledgments

The work is partly supported by the *Circle for the Promotion of Science and Engineering*, and *Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists*.

References

1. L. Baxter. *The complexity of unification.* PhD thesis, Department of Computer Science, University of Waterloo, 1977.
2. R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1996.
3. F. W. Burton and R. D. Cameron. Pattern matching with abstract data types. *J. Funct. Program.*, 3(2):171–190, 1993.
4. C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *GPCE*, pages 57–76, 2003.
5. K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha. DSL implementation in MetaOCaml, Template Haskell, and C++. In *Domain-Specific Program Generation*, pages 51–72, 2003.
6. O. de Moor and G. Sittampalam. Generic program transformation. In *Third International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 116–149, Braga, Portugal, September 1998. Springer-Verlag.
7. M. Fähndrich and J. Boyland. Statically checkable pattern abstractions. In *ICFP*, pages 75–84, 1997.
8. W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool support for the interactive derivation of formally correct functional programs. In *Journal of Universal Computer Science*, volume 9, pages 173–188, 2003.
9. R. Heckmann. A functional language for the specification of complex tree transformation. In *Proc. ESOP*, volume 300 of *LNCS*, pages 175–190, 1988.
10. G. P. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

11. W. Kahl. Basic pattern matching calculi: a fresh view on matching failure. In Y. Kameyama and P. J. Stuckey, editors, *FLOPS*, volume 2998 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2004.
12. B. Krieg-Brückner, J. Liu, H. Shi, and B. Wolff. Towards correct, efficient and reusable transformational developments. In M. Broy and S. Jähnichen, editors, *KORSO — Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *LNCS*, pages 270–284. Springer-Verlag, 1995.
13. J. Launchbury, S. Krstic, and T. E. Sauerwein. Zip fusion with hyperfunctions. available at http://www.cse.ogi.edu/ krstic/folds.pdf, 2000.
14. S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *POPL*, pages 333–343, 1995.
15. MetaML, Oct. 2000. Available online from `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.
16. MetaOCaml: A compiled, type-safe multi-stage programming language, 2003. Available online from `http://www.metaocaml.org/`.
17. M. Mohnen. Context patterns in haskell. In *Implementation of Functional Languages*, pages 41–57, 1996.
18. M. Mohnen. Context patterns, part II. In *Implementation of Functional Languages*, pages 338–357, 1997.
19. P. Palao-Gostanza, R. Pena, and M. Núñez. A new look to pattern matching in abstract data types. In *ICFP*, pages 110–121, 1996.
20. S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall, 1987.
21. S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, 2001.
22. T. Sheard and S. P. Jones. Notes on Template Haskell version 2, 2003.
23. T. Sheard and S. L. Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop*, pages 1–16, Pittsburgh, Pennsylvania, May 2002.
24. S. Thompson. Lawful functions and program verification in miranda. *Sci. Comput. Program.*, 13(2-3):181–218, 1990.
25. M. Tullsen. First class patterns. In E. Pontelli and V. S. Costa, editors, *PADL*, volume 1753 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2000.
26. E. Visser. Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *LNCS*, pages 357–362. Springer-Verlag, 2001.
27. J. Voigtländer. Concatenate, reverse and map vanish for free. In *ICFP*, pages 14–25, 2002.
28. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL*, pages 307–313, 1987.
29. T. Yokoyama, Z. Hu, and M. Takeichi. Deterministic second-order patterns. *Information Processing Letters*, 89(6):309–314, Mar. 2004.

A Proofs

Before proving Lemma 1, we prepare the following Lemma. This proof is the adaptation from Lemma 2 in [29].

Lemma 4. *If $P = \lambda x_1 \cdots x_l.\ f\ E_1 \cdots E_n$ is a $\mathcal{DHP}_3$ where f is a free variable, then there is at most a single match ϕ such that $\phi \vdash P \to T$.*

Proof. We assume that P, T, and $E_i (0 \leq i \leq n)$ are all in normal form. There is no match if T is not transformed into $\lambda x_1 \cdots x_l.\ T'$ by $\alpha\eta$-conversion. The match of a rule $f\ E_1 \cdots E_n \to T'$ should be in the form $\{f \mapsto \lambda y_1 \cdots y_n.\ B\}$. There is no loss of generality if we assume B is a normal form. Since free variables in each E_i are bounded in P by Definition 1.(iii), by definition of match the equation $(\lambda y_1 \cdots y_n.\ B)\ E_1 \cdots E_n =_{\alpha\beta\eta} T'$ should be satisfied. Let $\lambda z_i^1 \cdots z_i^{k_i}.\ B_i$ be $E_i (1 \leq i \leq n)$ in which B_i is not lambda abstraction. This just ensures that all the leading lambdas are accounted for by $z_i^1 \cdots z_i^{k_i}$. A term B is a result of replacing an instance of B_i — in which $z_i^1 \cdots z_i^{k_i}$ are instantiated and x_i's should not be instantiated — with $\phi_{h_i}\ (y_i\ z_i^1 \cdots z_i^{k_i})$ in T' where ϕ_{h_i} is a solution of matching B_i with the subterm of T' to replace. Since, by Definition 1, P is at most third-order and therefore B_i is at most first-order, and substitution ϕ_{h_i} is obtained deterministically for each subterms of T', if there exists. The domain of the substitution ϕ_{h_i} contains all the variables $z_i^1, \ldots, z_i^{k_i}$ by Definition 1 (iv). Therefore, variables $z_i^1, \ldots, z_i^{k_i}$ in $\phi_{h_i}\ (y_i\ z_i^1 \cdots z_i^{k_i})$ are deterministically instantiated. By Definition 1.(i), subterms $E_i\ (1 \leq i \leq m)$ contain free variables and if we leave any occurrences of instances of B_i in B, then $\lambda y_1 \cdots y_n.\ B$ will contain free variables. This results in generating illegal substitution containing free variables in its range. Instead, a term B should be obtained by full discharging; replacing all the occurrences of instances of B_i with $\phi_{h_i}\ (y_i\ z_i^1 \cdots z_i^{k_i})$ in T', i.e,

$$(\lambda y_1 \cdots y_n.\ B)\ (\lambda z_1^1 \cdots z_1^{k_1}.\ B_1) \cdots (\lambda z_n^1 \cdots z_n^{k_n}.\ B_i) = T'\ \wedge$$
$$\forall i.\ \{z_i^1 \mapsto \Box, \ldots, z_i^{z_i} \mapsto \Box\}\ B_i \not\trianglelefteq_{\Box} \{y_i \mapsto \Box, \ldots, y_n \mapsto \Box\}\ B.$$

If some free variables still occur in B after the discharging, this results in illegal substitution. Otherwise, by Definition 1.(ii), the order of this discharging does not affect the result of the match. Thus, the match is obtained deterministically.

Proof of Lemma 1
This proof is the adaptation from Theorem 3 in [29].

Proof. We use mathematical induction on the structure of the pattern.

- Case $(P = c\ E_1 \cdots E_m)$. There is no match if the corresponding term is not in the form of $c\ F_1 \cdots F_m$. Otherwise, the matching can be decomposed into m matchings $\phi_i, \Delta_i \vdash E_i \to F_i$ for $i = 1 \ldots m$. By the induction hypothesis, each match $\phi_i, \Delta_i \vdash E_i \to F_i$ is unique or there is no match in which case $\phi_i = \mathit{fail}$. Therefore $\phi, \Delta_1 \cup \cdots \cup \Delta_m \vdash P \to T$ is the unique match or there is no match if ϕ is *fail* where $\phi = \phi_1 \uplus \cdots \uplus \phi_m$.
- Case $(P = f\ E_1 \cdots E_m \wedge f \in \Delta)$. Similar to the first case.
- Case $(P = \lambda x_1 \cdots x_n.\ E)$. There is no match if the corresponding term cannot be transformed into $\lambda x_1 \cdots x_n.\ F$ by $\alpha\eta$-conversion. Otherwise, by induction hypothesis the match $\phi, \Delta \cup \{x_1, \ldots, x_n\} \vdash E \to F$ is unique or fail. Thus, the match $\phi, \Delta \vdash \lambda x_1 \cdots x_n.\ E \to \lambda x_1 \cdots x_n.\ F$ is unique or fail.
- Case $(P = f\ E_1 \cdots E_m \wedge f \notin \Delta)$. By Lemma 4, the match generated by the pattern is unique or there is no match.

Lemma 5 (Soundness). *The matching tree in Fig. 1 is sound.*

Proof. (Sketch) We use mathematical induction on the structure of the pattern. All the rest rules except FLEX1 and FLEX2 are easy to check correctness. Therefore, we focus on them.

For FLEX1, to ensure the correctness of *dreplaces*, we need to check

$$\begin{array}{l}\{f \mapsto \lambda v_1 \cdots v_m.\ T'\}\ (f\ (\lambda x_1^1 \cdots x_1^{k_1}.\ B_1) \cdots (\lambda x_m^1 \cdots x_m^{k_m}.\ B_m)) = T \\ \quad \textbf{where}\ T' = \mathit{dreplaces}\ [(B_1, v_1\ x_1^1 \cdots x_1^{k_1}), \ldots, (B_m, v_m\ x_m^1 \cdots x_m^{k_m})]\ T\end{array}$$

The calculation is as follows:

$$\begin{array}{ll} & \mathit{LHS} \\ = & (\lambda v_1 \cdots v_m.\ T')\ (\lambda x_1^1 \cdots x_1^{k_1}.\ B_1) \cdots (\lambda x_m^1 \cdots x_m^{k_m}.\ B_m) \\ = & \{v_1 \mapsto \lambda x_1^1 \cdots x_1^{k_1}.\ B_1, \ldots, v_m \mapsto \lambda x_m^1 \cdots x_m^{k_m}.\ B_m\}\ T' \\ = & \mathit{dreplaces}\ [(B_1, B_1), \ldots, (B_m, B_m)]\ T \\ = & \mathit{RHS}\end{array}$$

For FLEX2, to ensure the correctness of *replaceLT*, we need to check

$$\begin{array}{l}(\{f \mapsto \lambda v.\ T'\} \uplus (\phi \ominus \{x_1, \ldots, x_m\}))\ (f\ (\lambda x_1 \cdots x_m.\ B)) = T \\ \textbf{where}\ (T', \phi) = \mathit{replaceLT}\ (T, (B, v\ x_1 \cdots x_m)) \quad v \in \mathit{FV}(T')\end{array}$$

The calculation is as follows:

$$\begin{array}{ll} & \mathit{LHS} \\ = & (\phi \ominus \{x_1, \ldots, x_m\})\ ((\lambda v.\ T')\ (\lambda x_1 \cdots x_m.\ B)) \\ = & (\phi \ominus \{x_1, \ldots, x_m\})\ (\{v \mapsto \lambda x_1 \cdots x_m.\ B\}\ T') \\ = & (\phi \ominus \{x_1, \ldots, x_m\})\ T'' \\ & \quad \textbf{where}\ (T'', \phi) = \mathit{replaceLT}\ (T, (B, B)) \\ = & \mathit{RHS}\end{array}$$

□

Lemma 6 (Termination). *The matching algorithm using the matching tree in Fig. 1 terminates.*

Proof. Since, in matching tree, all the patterns in the premise is smaller than that of the conclusion, the proof of termination is settled down to the proof of termination of function *dreplaces* and *replaceLT*. In the function *dreplaces* the size of patterns are strictly decreasing in each calling, and in the function *replace* the size of the list of the second argument strictly decreasing. Function *replaceLT* calls *dr* and it traverses the given expressions at most once and terminates. Therefore, the matching algorithm terminates.

Proof of Lemma 2

Proof. By Lemma 1, 5, and 6, it holds. □

Proof of Lemma 3

Proof. By Lemma 5 and 6, the matching algorithm returns $\phi \vdash P \to T$. Recall that *replaceLT* aims to replace the instance of B with X. In the premise of FLEX2, $v \in FV(T')$ holds. Therefore, $\{f \mapsto \lambda v.\ T'\}$ must not be trivial. In the application case of function dr, if the recursive application of *replaceLT* to the function part returns the succeeded match, the argument part is not applied by *replaceLT*. Otherwise, $E_1 = E_1'$ and *replaceLT* is applied to E_2. This implies that the place to be replaced from B to X is at most one. This traversal is from top-down and left-to-right. Therefore, $\{f \mapsto \lambda v.\ T'\}$ is the maximum match with respect to the LT order. □

Lemma 7 (Efficiency). *The matching algorithm using the matching tree in Fig. 1 is linear time with respect to the size of the given term.*

Proof. (sketch) Function *dreplaces* is linear time with respect to the size of T and function *replaceLT* is linear time with respect to the size of T. Since both $\mathcal{DHP}_3$ and A_{LT} are linear patterns, to compute the composition of matches by operator $\uplus$, it does not need to check compatibility and it is computed in constant time. □

Proof of Theorem 1

Proof. By Lemma 2, 3, and 7, it holds □