A Matching Process Modulo a Theory of Categorical Products

Francoise Bellegarde

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-020

September, 1990

A Matching Process Modulo a Theory of Categorical Products

Françoise BELLEGARDE
Oregon Graduate Institute of Science and Technology
19600 NW von Neumann Dr.
Beaverton OR. 97006-1999
email: bellegar@cse.ogi.edu

Abstract

We present a matching algorithm modulo axioms of categorical products. This infinitary matching returns what we call selector-solved forms in which sets of equations have been simplified as much as possible. Although the selector-solved form is weaker than the fully solved form, it is sufficient for application to program transformation.

Introduction

Most program transformation systems consist of a catalogue of transformation rules that specify the schemes of the source S_1 and target S_2 programs together with a set of hypotheses H that must verify these programs. Program transformation can be viewed as schematic conditional rewriting systems. A transformation rule is a pair $\mathsf{S}_1 \rightarrow \mathsf{S}_2, (H)$ such that the source program scheme S_1 is equivalent to the target program scheme S_2 under the hypotheses H. In general, this equivalence is a congruence that allows us to transform programs in the following way: if a subpart t of the input program matches the source scheme S_1, and if the hypotheses H are satisfied, then the output program is the result of replacing t by the instantiation of the output scheme S_2.

The matching process has to instantiate functions, therefore this is at least a second-order matching as described in [9, 10, 7]. A way to avoid this is to use combinators, such as the composition operator, to express programs. If we use the composition, identity, projections and product to express programs, the axioms of the algebra of programs are those of categorical products. Then the matching process becomes first-order modulo these axioms. In this paper, we study the matching problem modulo the theory P of categorical products.

Following [16], the matching process is viewed as a step-by-step process that transforms a set of pairs of terms, simplifying these pairs until we get a normal-form, usually called the solved form: $x_1 = t_1, \cdots, x_n = t_n$, where $x_1, \cdots, x_n$ are variables.

A solved form gives the mapping of a matching substitution. However the matching modulo the theory P is infinitary (there can be infinitely many distinct solutions), which makes the process of finding a complete set of matches non terminating. The way to avoid this result is to simplify the set of pairs as much as possible. Here the normal-form is not always a solved form but is a set of pairs of terms that we call a selector-solved form: $s_1 = t_1, \cdots, s_n = t_n$. This form presents a generic solution. In a selector-solved form, different kinds of normal-forms coexist as in [4].

Most of these solutions appear to be insignificant for the purpose of program scheme recognition. For example, we do not want solutions that are functions in P with pairs as a result, because they do not correspond to functions in programs. Therefore the matching has to take some particular constraints [14] into account. These constraints can be expressed as transformation rules for the matching process. We will show that the constraints added to P for program scheme recognition give solutions that are always in solved form, which means that the constrained matching modulo P is finite.

We introduce the theory P in Section 2 and the matching process in Section 3. Section 4 examines some rules to constrain the matching problem for the purpose of program scheme recognition.

1 Preliminaries

This section contains the definitions and notations used in the paper.

Given a set V of variables, and a set F of function symbols, $T(F, V)$ denotes the free F-algebra over V. *Terms* are the elements of $T(F, V)$. V(t) denotes the set of variables of a term t. $T(F)$ denotes the set of ground terms on F, i.e., without variables.

Terms are considered as functions from the free monoid on the natural numbers without zero, N_+^* to $F \cup V$. The domain of t is called the set of occurrences or positions of t. Therefore, $t \mid p$ denotes the subterm of t at the position p. $t \mid \epsilon$ is t.

Substitutions σ are endomorphisms of $T(F, V)$ with a finite domain $D(\sigma)$. A substitution is denoted by its graph $(x_1 \mapsto t_1), \cdots, (x_n \mapsto t_n)$. $I(\sigma)$ is the set of the variables introduced by the substitution σ, i.e., $V(t_1) \cup \cdots \cup V(t_n)$.

Given two terms s and t, a substitution σ is a match from s to t if $\sigma(s) = t$.

Let $t[s]_p$ be the term t in which the subterm at position p has been replaced by s. Let $E = \{l_i = r_i\}_{i \in I}$ be a set of equations such that l_i and r_i for $i \in I$ are terms. One step of E-equality, denoted by $s \leftrightarrow_E t$, is defined by: an equation $l = r$ in E; a position p in s; and a match σ from l to the subterm of s at the position p. Here $t = s[\sigma(r)]_p$. The reflexive symmetric transitive closure is the E-equality denoted by $=_E$.

Given two terms s and t, a substitution σ is an E-match from s to t if $\sigma(s) =_E t$.

Let X be a subset of V. We define a quasi-order on substitutions $\leq_E$ by $\sigma \leq_E \sigma'[X]$ iff there exists a substitution ρ such that for all x in X, $\rho(\sigma(x)) =_E \sigma'(x)$. This quasi-order induces classically a partial order also noted $\leq_E$ on the quotient of the set of substitutions by the equivalence associated by $\leq_E$.

Let $M_E(s,t)$ be the set of the E-matches from s to t. We define the complete set of E-matches from s to t, denoted $CSM_E(s,t)$, away from a set of variables W by:

1. *Variable protection*: For all σ in $CSM_E(s,t)$, $D(\sigma) \subseteq V(s)$ and $I(\sigma) \cap (W - V(t)) = \emptyset$
2. *Correctness*: $CSM_E(s,t) \subseteq M_E(s,t)$
3. *Completeness*: $\forall \rho \in M_E(S,t), \exists \sigma \in CSM_E(s,t)$ such that $\sigma \leq_E \rho[V(s)]$
4. Moreover, the $CSM_E(s,t)$ is said to be *minimal*: when $\forall \sigma, \sigma' \in CSM_E(s,t)$, $\sigma \leq_E \sigma' \Rightarrow \sigma = \sigma'$

The variable protection condition is a technical restriction that allows us to separate the variables introduced by the match from the variable of s.

A *matching equation*, denoted by $s =_E t$ seeks a minimal $CSM_E(s,t)$ when possible. In our case, t will be a ground term. The notation $=_E$ refers only to the E-equality and does not reflect the asymmetry of the matching problem, but we use it for reasons we discuss below. A matching problem is a finite set of matching equations $\{s_1 =_E t_1, \cdots, s_n =_E t_n\}$ seeking $CSM_E(s_1,t_1) \cap \cdots \cap CSM_E(s_n,t_n)$.

2 The theory P: A first-order term language to represent schemes

Let us consider one of the well-known transformation rules for recursion removal. In [6, 10], this transformation rule is expressed by the rule $\mathsf{S}_1 \rightarrow \mathsf{S}_2, (H)$ where

$$\mathsf{S}_1 : f(x) = \textsf{ if } p(x) \textsf{ then } g(x) \textsf{ else } h(i(x), f(j(x))),$$

$$\mathsf{S}_2 : f(x) = \textsf{ if } p(x) \textsf{ then } g(x) \textsf{ else } w(i(x), j(x))$$

where

$$w(x,y) = \textsf{ if } p(y) \textsf{ then } h(x, g(y)) \textsf{ else } w(h(x, i(y)), j(y))$$

under the associativity of h:

$$\mathsf{H} : h(x, h(y,z)) = h(h(x,y), z)$$

In the rule, the terms express program schemes where the unknowns p, g, h, i, j stand for functions and x, y stand for data. This means that the terms are at least second-order.

Let us now express recursive definitions of functions with combinators: $\circ$ for the composition of two functions, Id for the identity function, $\langle -, - \rangle$ for the pairing of two functions, and Fst and Snd for the first and second projections. The symbol *quote*, denoted by '$-$, is the combinator K. It indicates constant functions. Such combinators exist in functional languages, for example in ML [17]. They are fundamental to

FP [1]. These combinators are the symbols of the theory of categorical products [15] that we call P. Suppose we have also a combinator Cond of arity 3 for conditionals. In such a combinatory framework the terms of the transformation rule become:

$$\mathsf{S}_1 : f = \mathsf{Cond}(p, g, h \circ \langle i, f \circ j\rangle),$$

$$\mathsf{S}_2 : f = \mathsf{Cond}(p, g, w \circ (h \circ \langle i, j\rangle)),$$

$$w = \mathsf{Cond}(p \circ \mathsf{Snd}, h \circ \langle \mathsf{Fst}, g \circ \mathsf{Snd}\rangle, w \circ \langle h \circ \langle \mathsf{Fst}, i \circ \mathsf{Snd}\rangle, j \circ \mathsf{Snd}\rangle)$$

and,

$$\mathsf{H} : h \circ \langle x, h \circ \langle y, z\rangle\rangle = h \circ \langle h \circ \langle x, y\rangle, z\rangle$$

This transformation scheme is cited in [13], where we can find a good catalogue of transformation schemes for recursion removal expressed in an FP-like formalism.

Now, variables stand only for functions. S_1, S_2, and H are defined by first-order terms.

Let us now consider a classical example. A function *reverse* reversing a list can be defined by the following recursive equation:

$$reverse(x) = \text{ if } null(x) \text{ then } nil \text{ else } append(reverse(tl(x)), cons(hd(x), nil))$$

In a combinatory framework, *reverse* becomes:

$$reverse = \mathsf{Cond}(null, `nil, append \circ \langle reverse \circ tl, cons \circ \langle hd, `nil\rangle\rangle)$$

How do we recognize that the scheme S_1 is applicable to the function reverse? If we keep the usual expressions and if we limit ourselves to simple functions, we can use a second-order matching. Such a second-order matching algorithm is proposed in [10]. If we choose the combinatory expressions presented above, the matching is now first-order modulo the theory P of categorical products. The theory P is elegant in that it creates a canonical (Confluent, Terminating and Interreduced) Term Rewriting System for P [5, 2].

$$\begin{aligned}
(f \circ g) \circ h &\rightarrow f \circ (g \circ h)\\
f \circ \mathsf{Id} &\rightarrow f\\
\mathsf{Id} \circ f &\rightarrow f\\
\mathsf{Fst} \circ \langle f, g\rangle &\rightarrow f\\
\mathsf{Snd} \circ \langle f, g\rangle &\rightarrow g\\
\langle \mathsf{Fst} \circ h, \mathsf{Snd} \circ h\rangle &\rightarrow h\\
`x \circ y &\rightarrow `x\\
\langle f, g\rangle \circ h &\rightarrow \langle f \circ h, g \circ h\rangle\\
\langle \mathsf{Fst}, \mathsf{Snd}\rangle &\rightarrow \mathsf{Id}
\end{aligned}$$

This Term Rewriting System allows us to normalize terms and to decide that two terms are P-equals by checking the equality of their normal-forms. We denote by $t \downarrow_P$, the normal-form of a term t using this system.

Let us now consider the matching problem modulo the theory P.

3 The matching problem modulo the theory P

3.1 Selector-solved forms

The matching problem modulo the theory P is infinitary, i.e., gives infinitely many independent solutions. For example we can easily find infinitely many independent solutions corresponding to the matching from S_1 to the expression of *reverse* given in Section 2:

$$p = null,\ g = \text{`}nil,\ j = tl,$$
$$h = append \circ \langle \mathsf{Snd}, \mathsf{Fst} \rangle,\ i = cons \circ \langle hd, \text{`}nil \rangle$$

but also

$$p = null,\ g = \text{`}nil,\ j = tl,$$
$$h = append \circ \langle \mathsf{Snd}, \mathsf{Fst} \circ \mathsf{Fst} \rangle,\ i = \langle cons \circ \langle hd, \text{`}nil \rangle, - \rangle$$

where $-$ stands for any term, and also

$$p = null,\ g = \text{`}nil,\ j = tl,$$
$$h = append \circ \langle \mathsf{Snd}, \mathsf{Fst} \circ \mathsf{Snd} \rangle,\ i = \langle -, cons \circ \langle hd, \text{`}nil \rangle \rangle$$

or

$$p = null,\ g = \text{`}nil,\ j = tl,$$
$$h = append \circ \langle \mathsf{Snd}, \mathsf{Fst} \circ \mathsf{Fst} \circ \mathsf{Fst} \rangle,$$
$$i = \langle \langle cons \circ \langle hd, \text{`}nil \rangle, - \rangle, - \rangle$$

and so on.

We can easily see why this happens. Let us consider a match from $x \circ y$ to t where x and y are variables. Suppose that x is substituted by $\mathsf{Fst} \circ (\mathsf{Snd} \circ \mathsf{Fst})$; then y must be $\langle \langle -, \langle t, - \rangle \rangle, - \rangle$. We know that all compositions of Fst and Snd are solutions for x. We define a subset of terms called *SELECTOR*, defined recursively as follows:

Definition 1 *Let S be a new set of variables. Usually, we denote the elements of S by an indexed identifier sel.*

1. $S \subseteq SELECTOR$
2. $\mathsf{Fst}, \mathsf{Snd}, \mathsf{Id} \in SELECTOR$
3. $\mathsf{Fst} \circ s, \mathsf{Snd} \circ s \in SELECTOR$ *if* $s \in SELECTOR - \{\mathsf{Id}\}$

A SELECTOR term is ground when it is constructed without SELECTOR variables.

Let sel be a $SELECTOR$ variable. Suppose we match $x \circ y$ to $a \circ b$ where x, y are variables and a, b are constants. Solutions of the match are also solutions of the set of equations: $\{x =_P sel, sel \circ y =_P a \circ b\}$ or $\{x =_P a \circ sel, sel \circ y =_P b\}$ or $\{x =_P a \circ (b \circ sel), sel \circ y =_P \mathsf{Id}\}$. These sets are selector-solved forms for the match. For application to program transformation, it is sufficient to have the selector-solved form, although it is weaker than the fully solved form. For the same purpose, G. Huet in [9] also introduced a weaker form of solved-form called "presolved forms" in higher-order unification processes.

Let A be the axiom of the associativity of $\circ$. We define a selector-solved form by:

Definition 2 *A selector-solved form for P is any finite set of equations*

$$\{s_1 =_P t_1, \cdots, s_n =_P t_n\}$$

such that $t_i \in T(F, V \cup S)$, and either $s_i \in V$ or $s_i =_A \varpi \circ y$ where $\varpi \in SELECTOR$, and $y \in V$.

For example the selector-solved form

$$\{p =_P null, g =_P {}^{\backprime}nil, j =_P tl,$$

$$h =_P append \circ \langle \mathsf{Snd}, sel_1\rangle, sel_1 \circ i =_P cons \circ \langle hd, {}^{\backprime}nil\rangle\}$$

gives a representation of an infinite family of solutions of the P-matching problem $\mathsf{S}_1 =_P reverse$.

3.2 Matching modulo D

We present an abstract view of the matching process as a set of rules for transforming a matching problem into an explicit representation of its solution, if such exists. This point of view was considered first by [8, 16]. Our rules are similar to the rules in [12].

P-match uses a set of transformation rules to process matching modulo a theory D with axioms:

$$\begin{aligned}
(f \circ g) \circ h &= f \circ (g \circ h)\\
f \circ \mathsf{Id} &= f\\
\mathsf{Id} \circ f &= f\\
\langle f, g\rangle \circ h &= \langle f \circ h, g \circ h\rangle\\
{}^{\backprime}x \circ y &= {}^{\backprime}x
\end{aligned}$$

A canonical Term Rewriting System for D is obtained by directing these equations from left to right.

A matching equation in the theory D is denoted by $s =_D t$ where $s \in T(F, V)$ and $t \in T(F)$. This matching is finite.

Using the canonical Term Rewriting System for D, we only consider normalized matching equations and normalized solved forms, i.e., all the terms are in D-normal-form.

to the D-equality. No confusion is possible between these two kinds of "equations" in our processes. In the rules **Mergedelete** and **Mergefail**, the condition $s =_D t$ and $s \neq_D t$ can be replaced by the syntactic equality $s = t$ and $s \neq t$ respectively because s and t are in D-normal form. The distinction between constant and variable in the Mutation rules are useful for performance and are made possible because all terms are in D-normal form.

These transformations are non deterministic. For example, one can apply either the rule **oo-Mutatel**, **oo-Mutate2**, or **Decompose**, which have the same left-hand side. Each option generates a possible solution.

Lemma 1 *Starting with $s =_D t$ and using the rules repeatedly until none is applicable results in a tree whose leaves are labeled by either $(\emptyset; fail)$, or $(\emptyset; \{x_1 =_D t_1, \cdots, x_n =_D t_n\})$ where $x_i \in V$. By removing the equations $new_i =_D t_i$, the set $x_1 =_D t_1, \cdots, x_m =_D t_m$ can be turned into a substitution D-match from s to t. In so doing, a complete set of D-matches from s to t is defined.*

Proof: Let us only sketch the steps of the proof.

- First, we group together rules having the same possible left-hand sides. For example, **oo-Mutatel**, **oo-Mutate2** and **Decompose** must belong to the same group. We show that each group of rules determines a set of equations whose set of solutions B is the same as the set of solutions A of the common left-hand side. This prove the *soundness* of the matching process. This is obvious in proving that $B \subseteq A$. The converse must take the axioms into account.

- Second, we show that the leaves of a transformation tree are either $(\emptyset; fail)$, or $(\emptyset; \{x_1 =_D t_1, \cdots, x_n =_D t_n\})$ where $x_i \in V$. This is easy, because the rules have been written for this purpose. In fact, all that we show is that no cases have been forgotten. This proves the *completeness*.

- Finally, it remains to prove that the process *terminates*. Let $\mid t \mid$ denote the size (number of symbols) of a term t. We define a well-founded ordering on equations by $s =_D t >_D s' =_D t'$ if $\mid s \mid + \mid t \mid\rangle \mid s' \mid + \mid t' \mid$. We compare the set E by the multiset extension of the ordering $\gg_D$. By looking at the rules **oo-Mutate2** and **opair-Mutate2**, we show that the number of new variables is bounded by the size of t. Each application of a transformation rule either enlarges the solved set Q which is bounded by the number of variables in s plus the number of new variables or the set E decreases by the well-founded multiset ordering $\gg_D$.

□

Obviously, the set of solutions is not minimal. However, because the set is finite, we can get a minimal solution by eliminating the redundancies.

Let the D-match algorithm be a set of transformation rules operating on pairs (E;Q) of sets of normalized matching equations, with E containing the equations yet to be solved and Q containing the partial normalized solution.

We denote by $E\{x \mapsto t\} \downarrow_A$ the set of equations E in which every occurrences of the variable x is replaced by the term t. Moreover all the terms in E are A-normalized.

A subset of V is a set of new variables, called $new, new_1, \cdots$, that are used during the matching process.

In the following, x always denotes a variable in V that can be a new variable for the process. new, new_1, new_2 denote new variables. c denotes a constant symbol. Notice that a term t in D-normal form with top symbol $\circ$ can only be of the form $c \circ v$ or $x \circ v$ where c is a constant symbol different from Id, x is a variable and, v, u are any term.

Let the transformation rules be:

Delete	$(\{s =_D s\} \cup E; Q) \Longrightarrow (E; Q)$.
Decompose	$(\{f(\vec{s}) =_D f(\vec{t})\} \cup E; Q)$ $\Longrightarrow (\{s_1 =_D t_1, \cdots, s_n =_D t_n\}) \cup E; Q)$.
Fail	$(\{f(\vec{s}) =_D g(\vec{t})\} \cup E; Q) \Longrightarrow (\emptyset; fail)$, if $f \neq g$ and $f \neq \circ$.
Eliminate	$(\{x =_D s\} \cup E; Q)$ $\Longrightarrow (E\{x \mapsto s\} \downarrow_D; \{x =_D s\} \cup Q\{x \mapsto s\} \downarrow_D)$ if x does not occurs in the left-hand sides in E.
Mergedelete	$(\{x =_D s, x =_D t\} \cup E; Q) \Longrightarrow (\{x =_D s\} \cup E; Q)$ if $s =_D t$.
Mergefail	$(\{x =_D s, x =_D t\} \cup E; Q) \Longrightarrow (\emptyset; fail)$ if $s, t \in T(F)$ and $s \neq_E t$.
oc-Mutate1	$(\{u \circ v =_D c\} \cup E; Q) \Longrightarrow (\{u =_D \mathsf{Id}, v =_D c\} \cup E; Q)$.
oc-Mutate2	$(\{u \circ v =_D c\} \cup E; Q) \Longrightarrow (\{u =_D c, v =_D \mathsf{Id}\} \cup E; Q)$.
oo-Mutate1	$(\{x \circ v =_D t_1 \circ t_2\} \cup E; Q) \Longrightarrow (\{x =_D \mathsf{Id}, v =_D t_1 \circ t_2\} \cup E; Q)$.
oo-Mutate2	$(\{x \circ v =_D t_1 \circ t_2\} \cup E; Q)$ $\Longrightarrow (\{x =_D t_1 \circ new, new \circ v =_D t_2\} \cup E; Q)$.
o‘-Mutate1	$(\{u \circ v =_D \text{‘}a\} \cup E; Q) \Longrightarrow (\{u =_D \mathsf{Id}, v =_D \text{‘}a\} \cup E; Q)$.
o‘-Mutate2	$(\{u \circ v =_D \text{‘}a\} \cup E; Q) \Longrightarrow (\{v =_D \text{‘}a\} \cup E; Q)$.
opair-Mutate1	$(\{x \circ u =_D \langle t_1, t_2 \rangle\} \cup E; Q)$ $\Longrightarrow (\{x =_D \mathsf{Id}, u =_D \langle t_1, t_2 \rangle\} \cup E; Q)$.
opair-Mutate2	$(\{x \circ u =_D \langle t_1, t_2 \rangle\} \cup E; Q)$ $\Longrightarrow (\{x =_D \langle new_1, new_2 \rangle, new_1 \circ u =_D t_1, new_2 \circ u =_D t_2 \rangle\} \cup E; Q)$.
opair-Mutate3	$\{c \circ u =_D \langle t_1, t_2 \rangle\} \cup E; Q) \Longrightarrow (\emptyset; fail)$

The rule Eliminate increases the solution set Q. It also allows us to eliminate the occurrences of the new variables. New variables are introduced as occurrences of terms t in equations of the form $x =_D t$ where $x \in V$ by the rules oo-Mutate2 and opair-Mutate2. The rule Eliminate achieves the solution for x when a solution for the new variables occurring in t is found. These equations with variables in the left-hand side are not really matching equations, but we keep the notation $=_D$ which refers

opair-Mutate1	$(\{x \circ u =_P \langle t_1, t_2\rangle\} \cup E; Q)$
	$\Longrightarrow (\{x =_P \mathsf{Id}, u =_P \langle t_1, t_2\rangle\} \cup E; Q)$.
opair-Mutate2	$(\{x \circ u =_P \langle t_1, t_2\rangle\} \cup E; Q)$
	$\Longrightarrow (\{x =_P \langle new1, new2\rangle, new1 \circ u =_P t_1, new2 \circ u =_P t_2\rangle\} \cup E; Q)$.
opair-Mutate3	$(\{x \circ u =_P \langle t_1, t_2\rangle\} \cup E; Q)$
	$\Longrightarrow (\{x =_P new_1 \circ sel, new1 \circ new_2 =_D \langle t_1, t_2\rangle, sel \circ u =_P new_2\} \cup E; Q)$.
o'-Mutate1	$(\{u \circ v =_P {}^{\backprime}a\} \cup E; Q) \Longrightarrow (\{u =_P \mathsf{Id}, v =_P {}^{\backprime}a\} \cup E; Q)$.
o'-Mutate2	$(\{u \circ v =_P {}^{\backprime}a\} \cup E; Q) \Longrightarrow (\{u =_P {}^{\backprime}a\} \cup E; Q)$.
o'-Mutate3	$(\{x \circ v =_P {}^{\backprime}a\} \cup E; Q) \Longrightarrow (\{x =_P sel, sel \circ v =_P {}^{\backprime}a\} \cup E; Q)$.

Once again, the rule opair-Mutate3 introduces a D-match equation.

Because of the axiom $\langle \mathsf{Fst} \circ f, \mathsf{Snd} \circ f\rangle \to f$, we have one more rule for a match from a term $\langle u, v\rangle$.

pair-Mutate	$(\{\langle u, v\rangle =_P t\} \cup E; Q) \Longrightarrow \{u =_P \mathsf{Fst} \circ t, v =_P \mathsf{Snd} \circ t\} \cup E; Q)$.

Let us consider now the case where we match a term $s = \Pi \circ (x\ o\ v)$ to a term t where Π is a ground selector term and v a term in P-normal form. Π is maximum in s. Other solutions can come from $\Pi \circ x$ being a selector of t in v. In this case x has to be a selector variable. Note that s is not itself in P-normal form. A $*$ indicates rules that do not follow the convention that all terms are in P-normal form.

*pi-variable	$(\{\Pi \circ (x \circ u) =_P t\} \cup E; Q) \Longrightarrow (\{x =_P sel, \Pi \circ (sel \circ u) =_P t\} \cup E; Q)$.

Lemma 2 *The above rules ensure soundness and completeness of the transformation.*

Proof: The soundness is obvious for each rule. For completeness, one must examine together all rules having the same left-hand sides and determine that all possible outcomes on the right-hand side have been considered. For example Decompose, oo-Mutate1, oo-Mutate2, and oo-Mutate3, having $x \circ v$ and $c' \circ t$ on the left-hand side, must be grouped.

Let us show how we prove completeness for this particular case. Because of the canonicity of P, any proof of $\sigma(x \circ v) =_P c \circ t$ corresponds to a rewrite proof of the form $\sigma(x \circ v) \downarrow_P = c \circ t$. The obvious solution is given by Decompose; oo-Mutate1 uses the left-identity; and oo-Mutate2 uses the associativity. The distributivity cannot yield $c \circ t$. oo-Mutate3 handles the projections. The main point is to prove that $new_1 \circ new_2 =_D c' \circ t$ has the same solution as $new_1 \circ new_2 =_P c' \circ t$ when $new_1 \neq_P u \circ \varpi$ where $\varpi \in SELECTOR$. This is done by induction on the structure of the substitution for new_1.

Other cases yield the same kind of proof. □

3.3 Mutation rules

Let us now consider a set of transformation rules for P-matching. Let us recall that we introduced the terms called $SELECTORS$ to define infinitely many independent solutions. S is a set of variables of the sort $SELECTOR$.

Using the canonical Term Rewriting System for P, we only consider normalized matching equations and normalized selector-solved forms, i.e., all the terms are in P normal-form. In the following, x always denotes a variable in V, which can be a new variable designed for the process. sel, sel_i denotes SELECTOR variables in S. new, new_i denotes new variables. c denotes a constant symbol. We still have the rule **Decompose**:

Decompose	$(\{f(\vec{s}) =_P f(\vec{t})\} \cup E; Q) \Longrightarrow (\{s_1 =_P t_1, \cdots, s_n =_P t_n\}) \cup E; Q).$

Let us now consider the match from a term s with a top symbol $s \mid \epsilon = \circ$ to a term t. In a term $s = u \circ v$ in P-normal-form u can only be a variable x or a constant c.

oc-Mutate1	$(\{x \circ v =_P c'\} \cup E; Q) \Longrightarrow (\{x =_P \mathsf{Id}, v =_P c'\} \cup E; Q)$
oc-Mutate2	$(\{u \circ v =_P c'\} \cup E; Q) \Longrightarrow (\{u =_P c', v =_P \mathsf{Id}\} \cup E; Q).$
oc-Mutate3	$(\{x \circ v =_P c'\} \cup E; Q) \Longrightarrow (\{x = sel, sel \circ v =_P c'\} \cup E; Q).$
oc-Mutate4	$(\{x \circ v =_P c'\} \cup E; Q) \Longrightarrow (\{x = c' \circ sel, sel \circ v =_P \mathsf{Id}\} \cup E; Q).$

The rules **oc-Mutate3** and **oc-Mutate4** introduce a variable of the sort $SELECTOR$ and an equation of the form $sel \circ v =_P t$ to prepare the eventual presentation of an infinite family of solutions. The above set of rules does not cover solutions that are possible when u in $s = u \circ v$ is a selector. We will include this case inside a more general case where $s \mid \epsilon = \circ$ and t is any term.

Let us now consider the match from a term s with a top symbol $s \mid \epsilon = \circ$ to a term t with a top symbol $t \mid \epsilon = \circ$.

oo-Mutate1 $(\{x \circ v =_P c' \circ t\} \cup E; Q)$
$\Longrightarrow (\{x =_P \mathsf{Id}, v =_P c' \circ t\} \cup E; Q).$
oo-Mutate2 $(\{x \circ v =_P c' \circ t\} \cup E; Q)$
$\Longrightarrow (\{x =_P c' \circ new, new \circ v =_P t\} \cup E; Q)$.
oo-Mutate3 $(\{x \circ v =_P c' \circ t\} \cup E; Q)$
$\Longrightarrow (\{x =_P new_1 \circ sel, sel \circ v =_P new_2, new_1 \circ new_2 =_D c' \circ t\} \cup E; Q\}).$

Note that the rule **oo-Mutate3** introduces a D-match equation $new_1 \circ new_2 =_D c' \circ t$, which will be transformed as indicated by Lemma 1.

Let us now consider the match from a term s with a top symbol $s \mid \epsilon = \circ$ to a term t with a top symbol $t \mid \epsilon \in \{', \langle -, - \rangle\}$.

3.4 Simplification of matching equations with SELECTOR variables

An equation $\varpi \circ (sel \circ v) =_P t$ where ϖ is a selector term, and v is not reduced to a variable is simplified by the following rules. π denotes one of the selectors Fst or Snd.

Projection1	$(\{sel \circ f(\vec{v}) =_P \pi\} \cup E; Q)$
$\Longrightarrow (\{f(\vec{v}) =_P \mathsf{Id}\} \cup E\{sel \mapsto \pi\}\downarrow_P; Q\{sel \mapsto \pi\}\downarrow_P\}$	
Projection2	$(\{sel \circ f(\vec{v}) =_P \pi \circ t\} \cup E; Q)$
$\Longrightarrow (\{sel_2 \circ f(\vec{v}) =_P t\} \cup E\{sel \mapsto \pi \circ sel_2\}\downarrow_P; Q\{sel \mapsto \pi \circ sel_2\}\downarrow_P\}$	
*Reducpairl	$(\{(\varpi \circ sel) \circ \langle u,v\rangle =_P t\} \cup E; Q)$
$\Longrightarrow (\{(\varpi \circ sel') \circ u =_P t\} \cup E\{sel \mapsto sel' \circ \mathsf{Fst}\}\downarrow_P; Q\{sel \mapsto sel' \circ \mathsf{Fst}\}\downarrow_P)$	
*Reducpair2	$(\{(\varpi \circ sel) \circ \langle u,v\rangle =_P t\} \cup E; Q)$
$\Longrightarrow (\{(\varpi \circ sel') \circ v =_P t\} \cup E\{sel \mapsto sel' \circ \mathsf{Snd}\}\downarrow_P; Q\{sel \mapsto sel' \circ \mathsf{Snd}\}\downarrow_P)$	
*Reducid	$(\{(\varpi \circ sel) \circ f(\vec{v}) =_P t\} \cup E; Q)$
$\Longrightarrow (\varpi \circ f(\vec{v}) =_P t\} \cup E\{sel \mapsto \mathsf{Id}\}\downarrow_P; Q\{sel \mapsto \mathsf{Id}\}\downarrow_P)$	
o-Reduc	$(\{\varpi \circ (sel \circ (x \circ u)) =_P t\} \cup E; Q)$
$\Longrightarrow (\{x =_P sel_2, \varpi \circ (sel \circ (sel_2 \circ u)) =_P t\} \cup E; Q)$.	

Lemma 3 *Starting with* $(\{sel \circ u =_P t\}; \emptyset)$, *and using the rules of Section 3.4 repeatedly until none is applicable results in a tree whose leaves are labeled by* $\{s_1 =_P t_1, \cdots, s_n =_P t_n\}; \{s'_1 =_P t'_1, \cdots, s'_m =_P t'_m\}$ *where* t_i *is a term in* $T(F,V)$ *and* $s'_i =_A \varpi \circ x$ *where* $\varpi \in SELECTOR$.

3.5 Delete and Merge rules

Let us now consider the deletion rules.

Delete	$(\{s =_P s\} \cup E; Q) \Longrightarrow (E; Q)$.
Fail	$(\{f(\vec{s}) =_P g(\vec{t})\} \cup E; Q) \Longrightarrow (\emptyset, fail)$
if $f \neq g$ and, f is a constant or $f =$ '.	
Eliminate	$(\{x =_P s\} \cup E; Q) \Longrightarrow (E\{x \mapsto s\}\downarrow_P; \{x =_P s\} \cup Q\{x \mapsto s\}\downarrow_P)$
if x is a variable which is not the left-hand side of another equation in E.	

Let us finally consider the merge rules. A new problem arises when we want to merge $x =_P s$ with $x =_P t$. If $s \neq t$ and if s or t contains variables of the sort $SELECTOR$, there is not a failure. Therefore we must find a substitution σ of these variables by terms of the sort $SELECTOR$ such that $\sigma(s) =_P \sigma(t)$. This restricted P-unification to $SELECTOR$ substitutions is introduced by unification equations denoted by $s =_U t$.

Merge	$(\{x =_P s, x =_P t\} \cup E; Q) \Longrightarrow (\{x =_P s, s =_P t\} \cup E; Q)$,
if $s, t \in T(F)$, $0 \leq \lvert s\rvert \leq \lvert t\rvert$.	
*Mergesel1	$(\{\varpi \circ x =_P s, x =_P t\} \cup E; Q) \Longrightarrow (\{\varpi \circ t =_P s\} \cup E; Q)$,
$s, t \in T(F)$.	
*Mergesel2	$(\{\varpi \circ x =_P s, \varpi \circ x =_P t\} \cup E; Q) \Longrightarrow (\{\varpi \circ x =_P s, s =_P t\} \cup E; Q)$,
$s, t \in T(F)$, $0 \leq \lvert s\rvert \leq \lvert t\rvert$.	
Mergeunif	$(\{x =_P s, x =_P t\} \cup E; Q) \Longrightarrow (\{x =_P s, s =_U t\} \cup E; Q)$
if $s, t \in T(F, S)$ and $s \neq t$.	

It remains to solve the restricted unification equations $s =_U t$.

Lemma 4 *The terms with variables of the sort SELECTOR in a restricted unification equation are P-equals to a term of the form $k \circ (\mathit{sel} \circ \Pi)$ where Π is a ground selector term and k is a ground term.*

Proof: A very informal argument is that these terms come from the equations $x = \mathit{new} \circ \mathit{sel}$ introduced by the mutation rules, thus *new* which is solution of a D-equation is instantiated by a ground term. Morever, any variable of the sort SELECTOR can be instantiated by SELECTOR terms with at least one SELECTOR variable (rules of Section 3.4). □

As a consequence *the restricted unification is an easy finite process.* We do not give the rules here for lack of space. They look like the rules that [12] uses to describe E-unification procedures except that variables are of a restrictive sort $SELECTOR$, i.e., terms belong to $T(F, S)$, and terms with variables are P-equals to terms of the restrictive form $k \circ (\mathit{sel} \circ \Pi)$.

3.6 Soundness, completness, and termination

Theorem 1 *Starting with $s =_P t$ and using all the rules of Section 3 repeatedly until none is applicable results in a tree whose leaves are labeled by either $(\emptyset; \mathit{fail})$ or $(\{s_1 =_P u_1, \cdots, s_m =_P u_m\}; \{x_1 =_P t_1, \cdots, x_n =_P t_n\})$ where $x_i \in V$, $s_i =_A \varpi \circ y$ where $\varpi \in SELECTOR$, and $y \in V$.*

Proof: Soundness and completeness are proved using the Lemmas. To prove the termination, we define a well-founded ordering on equations by $s =_A t >_P s' =_{A'} t'$ if $A \succ A'$ or $A = A'$ and $Max(\lvert s\rvert, \lvert t\rvert) \rangle Max(\lvert s'\rvert, \lvert t'\rvert)$ or $A = A'$ and $Max(\lvert s\rvert, \lvert t\rvert) = Max(\lvert s'\rvert, \lvert t'\rvert)$ and $nv(s) > nv(s')$, where $\succ$ is defined on $\{P, D, U\}$ by $P \succ D$ and $P \succ U$, and where $nv(s)$ is the number of variables of V occurring in s. We compare the set E by the multiset extension $\gg_P$. Now the proof is the same as for Lemma 1. □

Let us come back to our example in Section 2. We want to match S_1 which is expressed as:

$$f = \mathsf{Cond}(p, g, h \circ \langle i, f \circ j\rangle),$$

with

$$reverse = \mathsf{Cond}(null, \text{`}nil, append \circ \langle reverse \circ tl, cons \circ \langle hd, \text{`}nil\rangle\rangle)$$

We need to have the symbols $=$ and Cond in F. Then, by the rules Decompose and Eliminate we get the sets

$$(\{h \circ \langle i, f \circ j\rangle =_P append \circ \langle reverse \circ tl, cons \circ \langle hd, \text{`}nil\rangle\rangle\}; \\ \{f =_P reverse, p =_P null, g =_P \text{`}nil\})$$

Processing the unique equation in E returns 26 sets of distinct families of equations.

4 Constrained matching for scheme recognition

The trouble with transformation rules is that they often have implicit auxiliary syntactical constraints. For example, the scheme $\mathsf{S}_1 \rightarrow \mathsf{S}_2, (H)$ in the combinatory form assumes that the defined recursive variable symbol f does not occur in the function p, g, h, i, j. The validation of the scheme by fixpoint induction uses this fact, which actually is implicit for all the schemes for recursion removal proposed in [13].

To take care of this particular constraint, let us define subset R of V for the set of recursive variables, and let us add two transformation rules to the matching process:

f-constraint1	$(E; \{f =_P c, x =_P t\} \cup Q) \Longrightarrow (\emptyset; fail)$, if $f \in R$ occurs in t
f-constraint2	$(\{s \circ x =_P t\} \cup E; \{f =_P c\} \cup Q) \Longrightarrow (\emptyset; fail)$, if $f \in R$ occurs in t

For the match of S_1 with *reverse* which returns 26 solutions without constraints, the number of solutions fall down to 6. For example the following solution:

$$(\{sel_1 \circ i =_P \text{`}nil, sel_2 \circ i =_P hd, sel_3 \circ i =_P reverse \circ tl\}; \\ \{h =_P append \circ \langle sel_3 \circ \mathsf{Fst}, cons \circ \langle sel_2 \circ \mathsf{Fst}, sel_1 \circ \mathsf{Fst}\rangle\rangle \\ f =_P reverse, p =_P null, g =_P \text{`}nil\})$$

is eliminated, using f-constraint2, and the solution:

$$(\{sel_1 \circ i =_P \text{`}nil, sel_2 \circ i =_P hd, sel_3 \circ i =_P tl\}; \\ \{h =_P append \circ \langle reverse \circ (sel_3 \circ \mathsf{Fst}), cons \circ \langle sel_2 \circ \mathsf{Fst}, sel_1 \circ \mathsf{Fst}\rangle\rangle \\ f =_P reverse, p =_P null, g =_P \text{`}nil\})$$

is eliminated using f-constraint1.

Now, by using the combinator $\langle -, - \rangle$ to express the programs, we can have functions of the type $A \mapsto C \times D$. It may be, however that the programming language does not allow us to write functions of this type, which are considered not to be functions at all. In this case, we must add another constraint to the matching by the following rule:

*pair-constraint1	$(E; \{x =_P \langle u, v\rangle\} \cup Q)$
$\Longrightarrow (\emptyset; fail)$	
*pair-constraint2	$(\{\varpi \circ (sel \circ x) =_P t\} \cup E; Q)$
$\Longrightarrow (\{\varpi \circ x =_P t\} \cup E\{sel \mapsto \mathsf{Id}\} \downarrow_P; Q\{sel \mapsto \mathsf{Id}\} \downarrow_P)$	

For the match of S_1 with *reverse* the number of solutions decreases to 3 with pair-constraint1 and pair-constraint2. For example the solution:

$$(\{sel_1 \circ i =_P \text{`}nil, sel_2 \circ i =_P hd\};$$
$$\{h =_P append \circ \langle \mathsf{Snd}, cons \circ \langle sel_2 \circ \mathsf{Fst}, \mathsf{Fst}, sel_1 \circ \mathsf{Fst}\rangle\rangle,$$
$$j =_P tl, f =_P reverse, p =_P null, g =_P \text{`}nil\})$$

is eliminated. Moreover the solutions are in solved-form ($E = \emptyset$). We get the following result:

Corollary 1 *Starting with $s =_P t$ and using all the rules given in the paper repeatedly until none is applicable results in a tree whose leaves are labeled by either $(\emptyset; fail)$ or $(\emptyset; \{x_1 =_P t_1, \cdots, x_n =_P t_n\})$ where $x_i \in V$*

Proof: It is obvious that all the selector variables are suppressed, and the result follows by application of Theorem 1. □

The 3 solutions of the constrained matching are:

$$p = null, g = \text{`}nil, j = tl, h = append \circ \langle \mathsf{Snd}, \mathsf{Fst}\rangle, i = cons \circ \langle hd, \text{`}nil\rangle$$

$$p = null, g = \text{`}nil, j = tl, h = append \circ \langle \mathsf{Snd}, cons \circ \langle hd, \text{`}nil\rangle\rangle, i = \mathsf{Id}$$

$$p = null, g = \text{`}nil, j = tl, h = append \circ \langle \mathsf{Snd}, cons \circ \langle \mathsf{Fst}, \text{`}nil\rangle\rangle, i = hd$$

The correspondent solutions are directly given by a second-order matching (See [10]).

5 Conclusion

Our constrained matching process including the rule *pair-constraint* gives the same results as the second-order matching. It looks more complicated, but it is more general because we have a more general process without the pair-constraint rules. These rules restrict strongly the class of languages processed. For example, we might want to write a function such as $f = \langle reverse, length\rangle$ and be able to remove the recursion in the unfolded result. Even for a language that does not include the pairing, our process without *pair-constraint* is useful. Let us consider the example:

$$f(x, y) = \textbf{ if } null(x) \textbf{ then } nil \textbf{ else } append(unit(y), append(hd(x), g(tl(x), y)))$$

Using a second-order matching, the above definition does not match S_1, but the corresponding expression with combinators

$$f = \mathsf{Cond}(null \circ \mathsf{Fst}, \text{`}nil,$$
$$append \circ \langle unit \circ \mathsf{Snd}, append\langle hd \circ \mathsf{Fst}, g \circ \langle tl \circ \mathsf{Fst}, \mathsf{Snd}\rangle\rangle\rangle$$

P-matches S_1.

Moreover, we only want to retain a solution that satisfies the hypotheses. This can be viewed also as a special kind of constraint. However, constraints coming from the hypotheses can be difficult to handle. For example, some transformation schemes for recursion removal are such that a function h occurring in the target scheme does not appear in the source scheme. This function h has to be found by considering the hypothesis. In this case the hypothesis is an equation to solve in a theory T including P. Therefore the constraints introduced by the satisfaction of the hypothesis can provide a problem that is bigger than the original matching problem.

We have supposed that the operator Cond occurs only once at the top of the two terms of a matching equation, such that it immediately disappears by application of the rule Decompose. Consequently, it happens that we have to transform the term to match by using axioms on Cond (see [3, 11] for these axioms) in order to get Cond in such a position. One way to avoid that could be to take these axioms into account in the matching process itself.

I would like to thank Dick Kieburtz at OGI and, Pierre Lescanne, Helene Kirchner and Claude Kirchner at CRIN who provided me with useful discussions.

References

[1] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communication of the Association for Computing Machinery*, 21(8), 1978.

[2] F. Bellegarde. Rewriting systems on FP expressions to reduce the number of sequences yielded. *Science of Computer Programming*, 6, pages 11-34, North Holland, 1986.

[3] S. Bloom and R. Tindell. Varieties of *if* $\cdots$ *then* $\cdots$ *else* $\cdots$. In *SIAM Journal on Computing* 12(4) , pages 677-707, 1983.

[4] H. Comon and P. Lescanne. Equational Problems and Disunification. In *Journal of Computer Science*, Special issue on Unification. Part one, 7(3-4), pages 371-426, 1989.

[5] P. L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman, 1986.

[6] J. Darlington and R. Burstall. A System which automatically improves programs. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Standford, pages 479-484, 1973. Also: Acta Informatica, 6, pages 41-60, 1976.

[7] J. Hannan and D. Miller. Uses of Higher-Order Unification For Implementing Program Transformers. In *Proceedings of the Logic Programming Conference* MIT Press. Seattle, 1988.

[8] J. Herbrand. Sur la Théorie de la Démonstration,. In *Logical Writings*, W. Goldbach, ed., Cambridge, 1971.

[9] G. Huet. A Unification algorithm for typed lambda calculus. In *Theoritical Computer Science*, 1(1):27:57, 1973.

[10] G. Huet and B. Lang. Proving and Applying Program Transformations Expressed with Second-Order Patterns. *Acta Informatica* , 11, pages 31-55, Springer-Verlag, 1978.

[11] I. Guessarian and J. Meseguer. On the Axiomatization of "if-then-else". Internal Report, Center for the Study of Language and Information, CSLI-85-20, Standford, 1985.

[12] J.P. Jouannaud and C. Kirchner. Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification, Internal Report, Centre de Recherche en Informatique de Nancy, Nancy, 1989.

[13] R. B. Kieburz and J. Schultis. Transformations of FP program schemes. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture.* Association for Computing Machinery, 1981.

[14] C. Kirchner and H. Kirchner. Constrained Equational Reasoning. In *Proceedings of the ACM-SIGSAM International Symposium on Symbolic and Algebraic Computation*, pages 382-389, Portland, 1989.

[15] J. Lambek and P. J. Scott. Introduction to Higher-Order Categorical Logic. *Cambridge studies in advanced mathematics*, 7, Cambridge University Press, 1986.

[16] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions On Programming Languages And Systems*, 4(2):258-282, 1982.

[17] R. Milner. *A proposal for Standard ML.* Technical Report CSR-157-83, Computer Science Department, University of Edimburgh, 1983.